conference

proceedings

Fifth USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA
May 3-7, 1999

Sponsored by
The USENIX Association

The Advanced Computing Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $24 for members and $32 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past COOTS Proceedings

COOTS IV	1998	Santa Fe, New Mexico	$22/32
COOTS III	1997	Portland, Oregon	$20/30
COOTS II	1996	Toronto, Canada	$20/30
COOTS I	1995	Monterey, CA	$18/24

ISBN 1-880446-35-9

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Fifth USENIX Conference on

Object-Oriented Technologies and Systems

(COOTS '99)

May 3–7, 1999
San Diego, California, USA

Conference Organizers

Program Chair

Murthy Devarakonda, *IBM T.J. Watson Research Center*

Tutorial Program Chair

Douglas C. Schmidt, *Washington University*

Program Committee

Ken Arnold, *Sun Microsystems, Inc.*
Jennifer Hamilton, *Microsoft Corporation*
Doug Lea, *SUNY Oswego*
Gary Leavens, *Iowa State University*
Scott Meyers, *Software Development Consultant*
Ira Pohl, *University of California Santa Cruz*
Rajendra Raj, *Morgan Stanley & Company*
Douglas C. Schmidt, *Washington University*
Steve Vinoski, *IONA Technologies, Inc.*
Werner Vogels, *Cornell University*
Jim Waldo, *Sun Microsystems, Inc.*
Yi-Min Wang, *Microsoft Research*

Advanced Topics Workshop Chair

Joe Sventek, *Hewlett-Packard Labs*

The USENIX Association Staff

Contents

5th USENIX Conference on Object-Oriented Technologies and Systems

May 3–7, 1999
San Diego, California, USA

Thursday, May 6

Optimization
Session Chair: Werner Vogels, Cornell University

Programming in the Large
Session Chair: Joe Sventek, Hewlett-Packard Labs

Java Internet Measurements
Session Chair: Ken Arnold, Sun Microsystems, Inc.

Index of Authors

Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages

Gustaf Neumann and Uwe Zdun
Information Systems and Software Techniques
University of Essen, Germany
{gustaf.neumann,uwe.zdun}@uni-essen.de

Abstract

Scripting languages are designed for glueing software components together. Such languages provide features like dynamic extensibility and dynamic typing with automatic conversion that make them well suited for rapid application development. Although these features entail runtime penalties, modern CPUs are fast enough to execute even large applications in scripting languages efficiently.

Large applications typically entail complex program structures. Object-orientation offers the means to solve some of the problems caused by this complexity, but focuses only on entities up to the size of a single class. The object-oriented design community proposes design patterns as a solution for complex interactions that are poorly supported by current object-oriented programming languages. In order to use patterns in an application, their implementation has to be scattered over several classes. This fact makes patterns hard to locate in the actual code and complicates their maintenance in an application.

This paper presents a general approach to combine the ideas of scripting and object-orientation in a way that preserves the benefits of both of them. It describes the object-oriented scripting language XOTcl (*Extended* OTcl), which is equipped with several language functionalities that help in the implementation of design patterns. We introduce the filter approach which provides a novel, intuitive, and powerful language support for the instantiation of large program structures like design patterns.

1 Introduction

1.1 Scripting Languages

In applications, where the emphasis lays on the flexible reuse of components, scripting languages, like Tcl (Tool Command Language [25]), are very useful for a fast and high-quality development of software. The application development in scripting languages differs fundamentally from the development in systems programming languages [26] (like C, C++ or Java), where the whole system is developed in a single language. A scripting language follows a two-level approach, distinguishing between components (reusable software modules) and glueing code, which is used to combine the components according to the application needs. This two level approach leads to a rapid application development [26].

Scripting languages are typically interpreted and use a dynamic type system with automatic conversion. The application developer uses a single data type (strings) for the representation of all data. Therefore, the interfaces of all components fit together automatically and the components can be reused in unpredicted situations without change. The disadvantages of scripting languages are a loss in efficiency (e.g. for dynamic conversions and method lookup) and the lack of reliability properties of a static type system [18]. But these disadvantages can be compensated to a certain degree:

- For several application tasks, the loss of efficiency is not necessarily relevant, because the time critical code can be placed into components written in efficient systems programming languages. Only the code to control these components is kept in the highly flexible scripting language.

- Since the components are typically written in a language with a static type system the reliability argument applies only on the glue code, used to combine the components. To address these remaining problems we have integrated an assertion concept based on pre- and post-conditions and invariants (see [24]).

Since TCL is designed for glueing components together, it is equipped with appropriate functionalities, such as dynamic typing, dynamic extensibility and read/write introspection. Many object-oriented TCL-extensions do not support well these abilities in their language constructs. They integrate foreign concepts and syntactic elements (mostly adopted from C++) into TCL (see e.g. [15, 9]). Even less appropriate is the encouraged programming style in structured blocks and the corresponding rigid class concept, which sees classes as write-once, unchangeable templates for their instances. This concept is not compatible with the highly dynamic properties of the underlying scripting language.

1.2 Scripting and Object Orientation

The three most important benefits of object-orientation are encapsulation of data and operations, code reuse through inheritance, and polymorphism. These should help to reduce development time, to increase software reuse, to ease the maintenance of software and to solve many other problems.

But these claims are not undoubted: For example Hatton [11] argues that the non-locality problems of inheritance and polymorphism in languages, like C++, do not match the model of the human mind well.

Encapsulation lets us think about an object in isolation; this is related to the notion of manipulating something in short-term memory exclusively. Therefore, encapsulation fits the human reasoning. Since in scripting languages a form of code reuse is already provided through reusable components, the foremost reason for the use of object-orientation in a scripting language is the encapsulation. For that reason, the inheritance problem also seems less conflicting, because the inheritance is mainly used to structure the system and to put the components together properly. Inheritance in scripting applications normally does not lead to large and complex classes that are strongly dependent on each other.

Hatton [11] criticizes the polymorphism in C++ as damaging, because objects become more difficult to manipulate through the evolving non-locality in the structures. They involve a pattern-like matching of similar behavior in long-term memory. The string as an uniform and flexible interface instead of the use of polymorphism makes the objects easier to be put together. They get one unique behavior *and* one unique interface. They may be used in different situations differently, but the required knowledge about the object remains the same.

These arguments account for the glueing idea of the scripting language in the scope of a single class and its environment. This scope of a "programming in the small" is the strength of current object-oriented (language) concepts. Their weakness is the "programming in the large", where all components of a system have to be configured properly. The concepts only provide a small set of functionalities that work on structures larger than single classes, e.g. from languages like Java or C++ the following are known:

- virtual properties are used to define additional object- and class-properties,
- abstract classes specify formal interfaces and requirements for a set of classes,
- parametric class definitions are used for different data types on one class-layout.

Beneath such language constructs, methodical approaches, like frameworks, exist. Since they are coded using conventional language constructs the problems due to the language insufficiencies are not eliminated. The main insufficiency is that classes and objects are relatively small system-parts compared to an entire, complex system. Therefore, the wish for a language construct, which maps such a large structure to an instantiable entity of the programming language, arises.

1.3 OTcl – MIT Object Tcl

We believe OTCL [32] is an extraordinary object-oriented scripting language which supports several features for handling complexity. It preserves and extends the properties of TCL like introspection and dynamic extensibility. Therefore, we used OTCL as the starting point for the development of XOTCL.

In OTCL each object is associated with a class. Classes are ordered by the superclass relationship in a directed acyclic graph. The root of the class

hierarchy is the class `Object` that contains the methods available in all instances. A single object can be instantiated directly from this class. In OTcl classes are special objects with the purpose of creating and managing other objects. Classes can be created and destroyed dynamically like regular objects. Classes contain a repository of instance methods ("instprocs") for the associated objects and provide a superclass relationship that supports multiple inheritance.

Since a class is a special (managing) kind of object, it is managed by a special class called "meta-class" (which manages itself). The meta-class is used to create and to instantiate ordinary classes. By modifying meta-classes it is possible to change the behavior of the derived classes widely. All inter-object and inter-class relationships are completely dynamic and can be changed at arbitrary times with immediate effect.

2 Design Patterns in Scripting Languages

The complexity of many large applications is caused by the combination of numerous, often independently developed components, which have to work in concert. Typically, many classes are involved, with different kinds of non-trivial relationships, like inheritance, associations, and aggregations. Design patterns provide abstractions over reusable designs, that can typically be found in the "hot spots" [28] of software architectures. Patterns are designed to manage complexity by merging interdependent structures into one (abstract) design entity.

Design patters are considered increasingly often as reusable solutions for general problems. Specialized instances of design patterns can be used in a diversity of applications. Soukup [30] defines a pattern as follows:

> *"A pattern describes a situation in which several classes cooperate on a certain task and form a specific organization and communication pattern."*

Design patterns are collected in pattern catalogs [10, 6]. Typically, these catalogs contain general patterns, but there are also catalogs which collect domain specific patterns. In this paper, we see a design pattern as an abstract entity with normative, constructive and descriptive properties, that is identified in the design process and has to be preserved (with documentation and usage constraints) in the implementation as well.

2.1 Language Support for Design Patterns

Most efforts in the literature of design patterns collect and catalog patterns. These activities are very important, since they are the basis for new software architectures using design patterns. Soukup [30] remarks that this basic work is not yet ended.

Most authors present design patterns as guidelines for the design. When they are used in the design phase, the abstract pattern has to be transformed into a concrete implementation for each usage. A basic idea of this paper is to allow one to code a pattern once in an abstract way (e.g. for a pattern-library) and reuse it later in a specialized manner. The gained advantage is that patterns become (abstract) entities of the design process as well as of the implementation. This is similar to the use of the design process entities "object" and "class" which are also entities in object-oriented programming languages.

There are only a few efforts in the direction of language support for design patterns so far. We believe that one reason for this lack of support is due to the targeted languages. Conventional object-oriented programming languages, like C++, offer no support for reproduction of larger structures than classes (like design patterns). Therefore, it is nearly impossible to get a sufficient reproduction of such structures as an entity[1]. But there are more reasons [4], why language support for design patterns should be improved:

- *Traceability:* The pattern is scattered over the objects and, therefore, hard to locate and to trace in an implementation.
- *Self-Problem:* The implementation of several patterns requires forwarding of messages, e.g. an object A receives a message and forwards it to an object B. Once the message is forwarded, references to *self* refer to the delegated object B, rather than to the original receiver A. (known as the *self-problem* [17]).
- *Reusability:* The implementation of the pattern must be recoded for every use.

[1] Soukup [30] shows that some design patterns can be implemented as classes in C++ using `friend`, but Bosch [4] points out that these are only a few.

- *Implementation Overhead:* The pattern implementation requires several methods with only trivial behavior, e.g. methods solely defined for message forwarding.

Pree [28] identifies seven meta-patterns that define most of the patterns of Gamma et.al. [10]. This indicates that it is possible to find language constructs, which are able to represent all structures definable by these meta-patterns. In this work, we present the *filter* as such a language construct.

2.2 Language Support for Design Patterns in XOTcl

In the following sections, we describe the language support for design patterns we have developed. We introduce our ideas with examples from the language *Extended* OTcl (XOTcl, pronounced *exotickle*) which is an extension of OTcl, but we give no introduction to the language. Figure 1 shows the relationship between XOTcl and OTcl and lists important properties of OTcl.

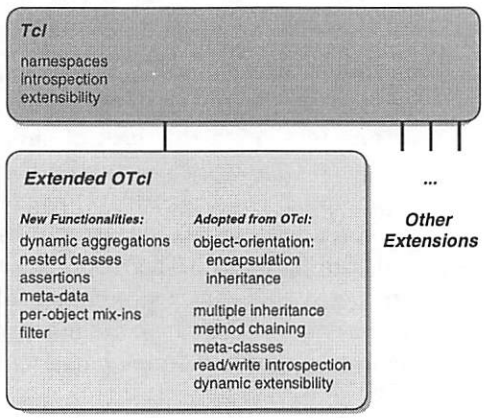


Figure 1: Language Extensions of XOTcl

Tcl and OTcl already have many properties that are very helpful for the implementation of patterns. Dynamic typing, as stated above, eases the management of highly generic structures. The definition of pattern parts as meta-classes makes them entities of the programming language and instantiable with the name of the pattern. Introspection allows self-awareness and adaptive programs, and simplifies the maintenance of relationships such as aggregations. Per-object specialization eases implementation of single objects with varying behavior, e.g. non-specializable singleton patterns.

In addition to the abilities of OTcl, we implemented in XOTcl new functionality specially targeted on complex software architectures and patterns. In particular, we added:

- *nested objects* based on Tcl's namespaces to (a) reduce the interference of independently developed program structures, (b) to support nested classes and (c) to provide dynamic aggregations of objects.
- *assertions* to reduce interface problems, to improve the reliability weakened by dynamic typing and, therefore, to ease the combination of many components,
- *meta-data* to provide self-documentation of objects and classes,
- *per-object mixins* as a flexible means to give an object access to several different addition-classes, which may be changed dynamically, and finally,
- *filters* as a means of abstractions over method invocations to implement patterns (see Section 2.3).

The first three extensions are variations of known concepts, which we have adopted in a dynamical and introspective fashion, the last two are both novel approaches. In [24, 33] we describe all these features in detail, in this paper we solely describe the filter approach.

2.3 The Filter Approach

We have pointed out that the realization of design patterns as entities is a valuable goal and that the object-oriented paradigm is not able to achieve this through classes alone. OTcl offers a means for the instantiation of large structures, like entire design patterns: the meta-classes. But in pure OTcl only a few patterns are instantiable this way (e.g. the abstract factory as in [33]), without suffering from the problems stated in Section 2.1. Typically, these patterns do not rely on a delegation or aggregation relationship.

Even though object-orientation orders program structures around the data, objects are characterized primarily by their behavior. Object-oriented programming style encourages the access of encapsulated data only through the methods of the object, since this allows data abstractions [31]. A method invocation can be interpreted as a message exchange between the calling and the called object. Therefore,

objects are only traceable at runtime through their message exchanges. At this point the filters can be applied, which are able to catch and manipulate all incoming and outgoing messages of an object.

> *A filter is a special instance method registered for a class C. Every time an object of class C receives a message, the filter is invoked automatically.*

A filter is implemented as an ordinary instance method (`instproc`) registered on a class. When the filter is registered, all messages to objects of this class must go through the filter, before they reach their destination object. The filter is free in what it does with the message, especially it can (a) pass (the potentially modified) message to other filters and finally to the object, or (b) it can redirect it to another destination, or (c) it can decide to handle the message solely.

The forward passing of messages is implemented as an extension of the `next` primitive of OTcl. `next` implements method chaining without explicit naming of the "mixin"-method. It mixes the same-named superclass methods into the current method (modeled after CLOS [5]). All classes are ordered in a linear next-path. At the point marked by a call to `next` the next shadowed method on this next-path is searched and, when it is found, it is mixed into the execution of the current method.

In XOTcl, a single class may have more than one filter. All the filters registered for a class form an ordered filter chain. Since every filter shadows all instance methods, `next` appears as a suitable mechanism to call the next filter in the chain. When all filters are worked through, the actual called method is invoked. By placement of the `next`-call, a filter defines if and at which point the remaining filters (and finally the actual method-chain) are invoked.

```
Class A
A instproc Filter-1 args {
  puts "pre-part of [self proc]"  ;# pre part
  next                            ;# next call
  puts "post-part of [self proc]" ;# post part
}
A filter Filter-1
A a1
a1 set x 1
```

This introductory example defines a single class and a single filter instproc. It registers the filter for the class using the `filter` instance method. An object `a1` is created. In the last line the predefined `set` method is invoked. Automatically the registered filter `Filter-1` of class `A` receives the message `set`. The filter instproc consists of three (optional) parts: The *pre-part* consists of the actions before the actual method is called, the `next` call invokes the message chaining, and the *post-part* contains the actions to be executed before the filter is left. In this example the pre- and post-parts are simple printing statements, but in general they may be filled with arbitrary XOTcl-statements. The distinction between the three parts is just a naming convention for explanation purposes.

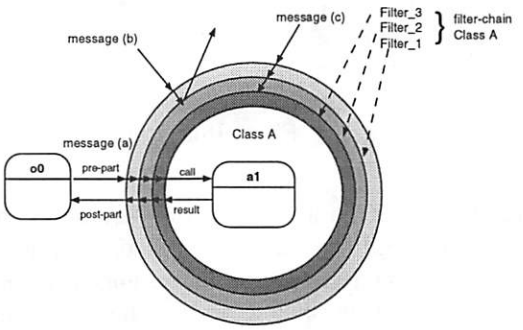


Figure 2: Cascaded Message Filtering

The following extension of the introductory example shows how to apply more than one filter, which are cascaded through `next` (see Figure 2). In this extended example a filter chain consisting of two filters is used. Again `next` forwards messages to the remaining filters in the chain or to the actual called method. The method `filter` registers the list of filters to be used.

```
A instproc Filter-2 args {
  puts "only a pre-part in [self proc]"
  next
}
A instproc Filter-3 args {
  next
  puts "only a post-part in [self proc]"
}
A filter {Filter-1 Filter-2 Filter-3}
```

When an instance `a1` of class `A` receives a message, like "`a1 set x 1`", it produces the following output. The `next`-call in the last filter `Filter-3` of the chain invokes the actual called method `set`.

```
pre-part of Filter-1
only a pre-part in Filter-2
only a post-part in Filter-3
post-part of Filter-1
```

The `filter` method can be used to remove filters dynamically as well. E.g. the filters `Filter-1` and `Filter-3` can be removed by:

```
A filter Filter-2
```

On each class the filters are invoked in the order specified by the `filter` instance method. To avoid circularities all filters which are currently active – that means that the current call is invoked directly

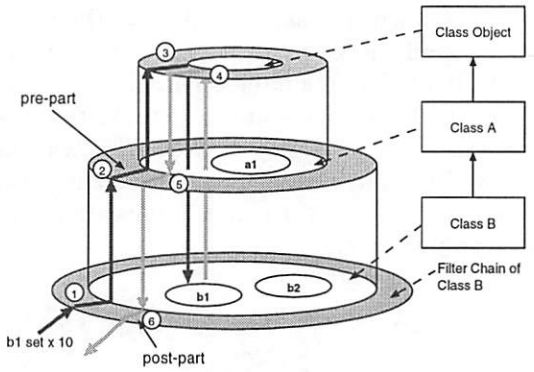


Figure 3: Filter Inheritance

or indirectly from a filter instproc – are temporarily left out of the filter chain. Filter chains can also be combined through (multiple) inheritance using `next`. Since filters are normal instprocs they may themselves be specialized through inheritance. When the end of the filter chain of the object's class is reached, the filter chains of the super-classes are invoked using the same precedence order as for inheritance.

This is demonstrated by the example displayed in Figure 3. `B` is a subclass of `A` with two instances `b1` and `b2`. Both instances are filtered with the chains registered on `B`, `A` and `Object`. The invocation `b1 set x 10` results in the next-path shown in Figure 3.

```
Class B -superclass A
B instproc Filter-B args {
  puts "entering method: [self proc]"
  next
}
B b1; B b2
B filter Filter-B
b1 set x 10
```

Filters have rich introspection mechanisms. Each class may be queried (using the introspection method `info filters`) what filters are currently installed. A filter method can obtain information about itself and its environment, and also about the calling and the called method. Examples are the name of the calling and the called method, the class where the filter is registered, etc. (see for details [24]). By using these introspection mechanisms filters can exploit various criteria in order to decide how to handle a message.

Often it is useful to add filters to an existing chain of filters. This can be achieved conveniently by the instproc `filterappend` defined for the top-level class `Object`. Therefore this method is inherited by all classes.

```
Object instproc filterappend f {
  [self] filter [concat [[self] info filters] $f]
}
A filterappend {Filter-2 Filter-3}
```

3 Language Support for Design Patterns using Filters

Now we present a systematic approach how filters can be used to implement design patterns. In general, filters are very flexible and well-suited for implementing patterns in various creative approaches.

3.1 Applying Filters on Meta-Patterns

In [28] Pree has identified meta-patterns as structures underlying several design pattern. They subdivide the pattern into a general, generic pattern-class (called *template*) and a class which serves as an anchor for the application specific details (called *hook*).

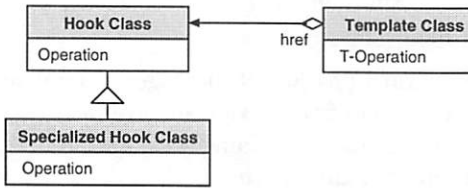


Figure 4: The 1:1 Meta-Pattern [28] with a Specialized Hook

Figure 4 shows a simple meta-pattern, that is based on a 1:1 association. It separates the specializations of a hook class from a template class. This is just an example pattern to give an idea of hook and template. It is obvious that many object-oriented structures, like several design patterns in [10, 6], are based upon this meta-structure.

The methods of the template implement the generic part of the structure and invoke the hook methods. The abstract hook forms a common interface for its specializations. The structure can be reused with different special hooks without changes to the template.

Filters are well-suited to implement meta-patterns. By using a filter all activities of a pattern can be treated in one entity (the filter instance method). Since all messages are directed to the filter the abstract tasks of the pattern can be separated from actual tasks of the application. But this alone would not be a reusable solution, since for every template class of every task, where the pattern could be used, a new filter method would have to be implemented. In order to achieve reusability we use a meta-class that provides the desired functionality. This meta-class may be stored in a library and can be reused every time a similar problem occurs.

The steps, to obtain a reusable and instantiable pattern based on filters from a pattern class diagram (e.g. Figure 4), are:

1. Find the hook and template classes.
2. Create a meta-class under the general name of the pattern.
3. Add a filter method to the meta-class, which performs all recurring tasks desired from the design pattern (especially the tasks of the template).
4. Add a constructor to the meta-class, which registers the filter on classes derived from the meta-class (and performs pattern specific initialization tasks).
5. Add additional methods to the meta-class (e.g. like registration of special hooks) to avoid hard-coding of pattern semantics in the filter method.

With slight adaptations this scheme is applicable on all patterns that rely on Pree's meta-patterns (e.g. most of the patterns in [10]). But nevertheless most other patterns, since they normally involve messages exchanges, are supportable by filters. A meta-class can be defined as a general solution for a large number of related problems. In order to use it, the application must derive a class from it (e.g. with the name of the template class) and concretize the application specific actions (that means the hook classes).

Now we show on a template for the 1:1 meta-pattern, how to apply the scheme in XOTcl. The first step is to define a meta-class. In XOTcl a meta-class is defined by referencing the meta-class `Class` as superclass of a newly defined class:

```
Class 1-1-Meta-Pattern -superclass Class
```

Secondly, a filter instproc must be defined:

```
1-1-Meta-Pattern instproc 1-1-Filter args {
  # filters actions
  # e.g. forwarding messages to the special hook
}
```

As the next step the constructor `init` registers the filter on the newly created class and performs other initialization tasks, like variable initialization, method declaration, etc.:

```
1-1-Meta-Pattern instproc init args {
  # initialization tasks
  [self] filterappend 1-1-Filter
}
```

For real applications the meta-class has to be extended with additional methods. In order to complete the implementation of the 1:1 meta-pattern a method, which stores a reference to the special hook on the object, has to be defined. Finally, the meta-pattern is instantiated to create a filtered template class.

```
1-1-Meta-Pattern FilteredTemplate
```

In order to provide the hook for the filter a special hook class and perhaps concretizations have to be created.

The presented scheme may be extended for more specialized patterns, e.g. a recursive pattern may require recursive registering of the filter. Sometimes it is useful (but not necessary) to apply a second filter (e.g. in patterns with a second referencing relationship, like mediator or observer in [10]).

3.2 Design Pattern Examples

The idea underlying meta-patterns splits patterns into two parts: the template and the hook. We have shown a scheme how to apply a filter if this division is possible. This section applies the scheme in order to implement three example patterns from [10].

3.2.1 The Adapter Pattern

The adapter pattern [10] converts the interface of a class into another interface that a client expects. Therefore, an adapter is a means to let classes cooperate despite of incompatible interfaces. As shown in Figure 5 the conventional solution is to forward the messages from `Adapter` to `Adaptee` by explicit calls. This approach entails that for every adapted method a new additional method must be defined in the adapter. This leads to an implementation overhead. Moreover, the solution's program code is neither reusable nor traceable.

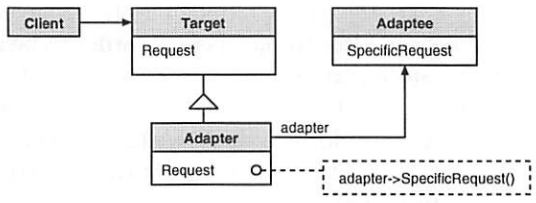


Figure 5: The Adapter Pattern [10]

The solution for the adapter problem presented below is based on filters and avoids these problems.

It is reusable and does not require the implementation overhead resulting from methods which are defined solely for the purpose of message forwarding. The forwarding is handled automatically by the next primitive in the filter method, no additional helper methods are needed.

By following the systematic steps presented above, we identify the template (here `Adapter`) and the hook (here `Adaptee`). The adapter pattern resembles the 1:1 meta-pattern of Section 3.1, but it has no special hooks. The desired actions of the template are to forward requests to specific requests. This will be handled by the filter. Firstly, we define a meta-class which replaces the pattern from the conventional design in Figure 5.

```
Class Adapter -superclass Class
```

A meta-class can be used to derive new classes that can access the instance methods of the meta-class. The derived classes are constructed with constructor of the meta-class. Next, we define the filter instance method:

```
Adapter instproc adapterFilter args {
  set r [[self] info calledproc]
  [self] instvar specificRequest adaptee \
    [list specificRequest($r) sr]
  if {[info exists sr]} {
    return [eval $adaptee $sr $args]
  }
  next
}
```

The `info calledproc` command returns the originally called method. This is the general request which is to be mapped to a specific request. The two variables `specificRequest` and `adaptee` are instance variables which are linked to the current scope by the primitive method `instvar`. The `specificRequest` for the called method is mapped to the variable `sr`. `adaptee` is the object which handles the specific requests. If there exists a mapping of the current request, the filter forwards the message to the associated method. Otherwise the message is not redirected, but passed further on by the filter along the next-path.

As the next step we have to define the constructor which adds the filter to the class. In order to be able to set the `specificRequest` and `adaptee` variables it is convenient to define instprocs for this purpose, which are defined for the derived classes. These instprocs are created dynamically by the constructor (the `init` instproc) of the meta-class:

```
Adapter instproc init args {
  [self] filterappend [self class]::adapterFilter
  next
  [self] instproc setRequest {r sr} {
    [self] set specificRequest($r) $sr
  }
  [self] instproc setAdaptee {a} {
    [self] set adaptee $a
  }
}
```

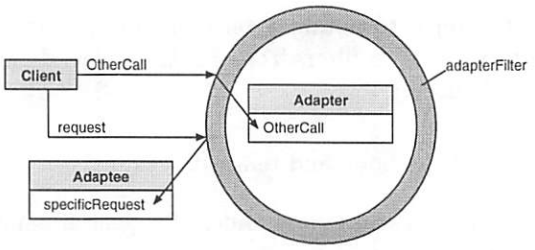


Figure 6: The Adapter Pattern Using Filters

Now the abstract pattern is converted into a meta-class, which can be used to derive classes with the behavior of the pattern: method invocations, which correspond to registered requests, are redirected to the adaptee object; all other invocations are passed unmodified to the object through the next-path (see Figure 6).

The solution in [10] suffers from the self-problem, since the originally called object of the adapter class is not the object which performs the desired task. This problem is not addressed by the filter solution presented above. A more sophisticated solution, which does not suffer from the self-problem, is to define the filter on the adaptee instead of the adapter. For the sake of simplicity we presented here the slightly simpler version.

A sample application of this pattern is a class which handles network connections. Derived classes, like `FTP`, `HTTP`, etc. allow one to handle specialized connections. All of them must implement a method `connect`. A method `discard` of the `Connection` class is able to close connections of all different kinds. Suppose a FTP connection routine from a library class with a different interface should be used. A filter adapter on basis of the defined meta-class can solve this problem elegantly. Firstly the interfaces of the related classes:

```
# interface of the library class
Class FTPLIB
FTPLIB instproc FTPLIB_connect args {...}

# the connection class
Class Connection
# an abstract connection method
Connection instproc connect args {...}
# the method to close a network connection
Connection instproc discard args {...}

... other class definitions, like HTTP
```

Now we derive a class FTP from the Adapter. The meta-class's constructor defines the two convenience

methods and registers the filter on the new class FTP automatically. Strictly speaking the convenience methods are not necessary, but they provide a simpler interface. The class FTP has a constructor that automatically creates an associated adaptee and provides the needed information for the filter though the convenience methods.

```
Adapter FTP -superclass Connection
FTP instproc init args {
  FTPLIB ftpAdaptee
  [self] setRequest connect FTPLIB_connect
  [self] setAdaptee ftpAdaptee
}
```

Finally, the FTP class can be used and is adapted automatically. Since only the method `connect` was a registered request, all `discard`-calls reach the `Connection` class.

```
FTP ftp1
ftp1 connect
...
ftp1 discard
```

This simple example can be extended with only a few more lines of code to provide more sophisticated adaptations (e.g. altering parameters, adapting to other objects, etc) without architectural redesign.

3.2.2 The Composite Pattern

A recursive pattern from [10] is the composite pattern, shown in Figure 7. The composite pattern helps to arrange objects in hierarchies with a unique interface type, called component. The objects are arranged in trees with two kinds of components: leafs and composites. Every composite can hold other components.

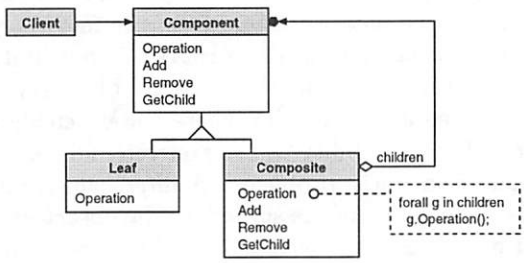


Figure 7: The Composite Pattern [10]

There are several disadvantages in the implementation of the pattern in [10]. The composite pattern structure contains dynamic object aggregation, what is not provided in C++. Therefore, the implementation lacks flexible mechanisms to handle and introspect the aggregation. An implementation overhead results from the necessity to define methods for management of the dynamic aggregation. Furthermore, the scattering of the pattern across several classes, leads to a mixing of application and pattern structures that reduces reusability.

The pattern (as presented in [10]) is not an abstract entity; therefore, it is hard to specialize and to reuse it. Also, it is not easy to find it in source code, if it is not well commented, and both description in the pattern classes and the runtime object structure are hard to introspect and not traceable.

In order to implement the pattern as a filter, we firstly identify its elements. The composite class forms the template, the component class the hook. The desired action of the template is to forward all messages to the aggregated objects recursively. The application specific actions are the concretizations that determine what these classes do with the messages. We create a meta-class:

```
Class Composite -superclass Class
Composite instproc addOperations args {...}
Composite instproc removeOperations args {...}
```

As a useful enhancement to the solution in [10], new operations are added and removed by `addOperations` and `removeOperations` (not to be confused with the methods for aggregation handling in Figure 7). Only registered operations will be forwarded to the objects in the composite patterns runtime structure.

All generic pattern tasks will be performed by a filter. It handles the forwarding to the components of a composite:

```
Composite instproc compositeFilter args {
  [[self] info class] instvar operations
  set r [[self] info calledproc]
  if {[info exists operations($r)]} {
    foreach object [[self] info children] {
      eval [self]::$object $r $args
    }
  }
  return [next]
}
```

In the composite filter firstly the request is compared to the operations in the `operations`-list. If the request is a registered operation, the message is forwarded to the child. Though children may be composites, this mechanism functions recursively on the entire structure, until the leaves are reached.

In order to register the filter on a new composite class automatically, we append it in the constructor of the meta-class:

```
Composite instproc init {args} {
  next
  [self] filterappend Composite::compositeFilter
}
```

Now we will show on an illustrative example that this single method handles *all* semantics of the pattern. As a sample application we will build up a simple graphic:

```
Class Graphic
Graphic instproc draw {} {...}
```

Different graphics objects can be defined on basis of the component type. For example we can define a Composite (Picture) with two leaves (Line and Rectangle):

```
Composite Picture -superclass Graphic
Class Line -superclass Graphic
Class Rectangle -superclass Graphic
```

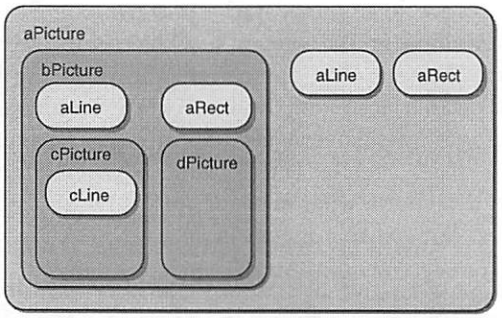


Figure 8: The Composite Object Structure

The graphic structure shown in Figure 8 can be constructed by:

```
Picture aPicture
Picture aPicture::bPicture
Line aPicture::aLine
Rectangle aPicture::aRect
Line aPicture::bPicture::aLine
Rectangle aPicture::bPicture::aRect
Picture aPicture::bPicture::cPicture
Picture aPicture::bPicture::dPicture
Line aPicture::bPicture::cPicture::cLine
```

An invocation of the `draw` method on a complex object, like:

```
Picture addOperations draw
```

registers the `draw` message for all the component objects in the structure. A call of `draw` draws the whole hierarchy:

```
aPicture draw
```

Note how simple and short it was to instantiate the sample application. Beneath the elimination of the problems mentioned above, compared to a solution of the picture application following [10], the filter solution is much shorter and easier to understand. It avoids complex structures that are connected in many ways, and removes the need for replicated code, since it takes the pattern semantics completely out of the application. Furthermore, the result is that the pattern is reusable as a program fragment (and may be put into a library of patterns) and not only as a design entity, which has to be recoded for every usage.

To map the recursive structure of the pattern a more general solution, in which each composite class gets recursively its own filter instproc, is easily achievable. This would allow one to specialize the filters behavior for certain branches of the structure (e.g. in order to fade out parts of the picture) or to store different composites, components or other classes in the pattern structure (what is possible to certain degree in the solution presented).

3.2.3 The Observer Pattern

The observer pattern presented in this section fulfills the task of informing a set of depending objects ("observers") of state changes in one or more observed objects ("subjects"). This problem is well known and often addressed, e.g. by the publisher subscriber pattern [6] or Model-View-Controller [14]. Figure 9 shows the observer design pattern as presented in [10].

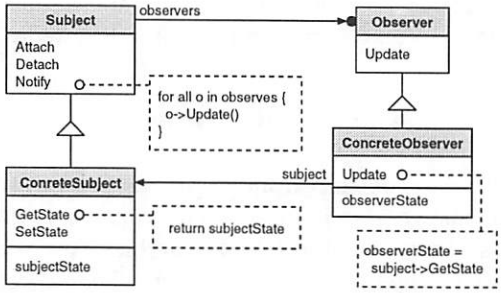


Figure 9: The Observer Pattern [10]

Bosch [3] identifies the problem that the traceability of the pattern suffers from the fact that the methods `attach`, `detach` and `notify` do not build up a conceptual entity and that the calls of `notify` must be inserted at every point where a state change occurs. The reusability of the concrete subjects also suffers from these problems. A filter, which directs all state changes of the subject to the observers does not have these problems and provides a reusable solution.

In order to implement an observer pattern based on filters we create meta-classes for the observer and the subjects. The subjects are structured as nested class to preserve the unity of the pattern:

```
Class Observer -superclass Class
Class Observer::Subject -superclass Class
```

In this example we only handle the relationship between subject (as template) and observer (as hook) by a filter. In a more sophisticated solution the

second referencing relationship between concrete observer and concrete subject may also be replaced by a filter. Now we can define a filter which handles the notification:

```
Observer::Subject instproc notifyFilter args {
  set r [[self] info calledproc]
  [self] instvar preObservers postObservers \
        [list preObservers($r)  preObs] \
        [list postObservers($r) postObs]
  if {[info exists preObs]} {
    foreach o $preObs {$o update [self] $args}
  }
  set result [next]
  if {[info exists postObs]} {
    foreach o $postObs {$o update [self] $args}
  }
  return $result
}
```

Observers are registered with the `attach` and `detach` methods. As a special feature we allow both pre- and post-observers to be registered. When the filter method is invoked, firstly all registered pre-observers are informed, then the actual method is invoked and then all post-observers are informed. Finally, the filter returns the result of the called method.

The trivial methods to register or unregister observers (here: `attachPre`, `attachPost`, `detachPre` and `detachPost`) are created by the constructor `init` on all instantiated classes, so that their objects can reach them as `instproc`'s (not presented here).

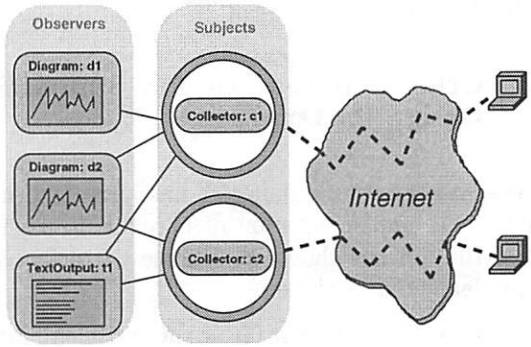


Figure 10: Observer Example

We demonstrate the usage of the abstract observer pattern by an example of a network monitor which observes a set of connections and maintains several views on these (e.g. a diagram and a textual output). In the implementation the class `Pinger` encapsulates the view and collector classes, the collectors are treated as subjects of the observer:

```
Class Pinger
Observer::Subject Pinger::Collector
Observer Pinger::Diagram
Observer Pinger::TextOutput
```

The `Collector` starts the observation of the network connection in its constructor, e.g.:

```
Pinger::Collector instproc init args {
  set hostName 132.252.180.67
  set f [open "| /bin/ping $hostName" r]
  fconfigure $f -blocking false
  fileevent $f readable "[self] ping \[gets $f\]"
}
```

The operation `ping` is the network event, which must be handled by the collector. Since the collector is a concrete subject it needs a method (`getResponse`) which is invoked by the observers to get its current state:

```
Pinger::Collector instproc ping {string} {...}
Pinger::Collector instproc getResponse {} {...}
```

The two observers must concretize their `update` methods. Both must catch the actual state of the subject using `getResponse` and then they will update their presentation. The text output presentation may look like:

```
Pinger::TextOutput instproc update {subject args} {
  set response [$subject getResponse]
  puts "PINGER: $subject --- $response"
}
```

For concrete applications the classes must be instantiated. Here are two collectors, some observers and some attachments:

```
Pinger::Collector c1
Pinger::Collector c2
Pinger::Diagram d1
Pinger::Diagram d2
Pinger::TextOutput t1

c1 attachPre ping d1 d2
c1 attachPost ping d2 t1
c2 attachPost ping t1 d2
```

This attaches the diagrams and the text output to the collectors `c1` and `c2` as pre- and as post-observers, as shown in Figure 10.

4 Related work

There are many other concepts with names containing the word "filter" (e.g. in the area of mobile/distributed computing [27, 16]). The composition filter model [1] introduces the idea of a higher-level object interaction model through abstract communication types (ACTs). Besides such basic ideas of a means to change, redirect, or otherwise affect messages, we have not found an approach with comparable properties like filters (as user-defined methods, mixin of filter chains, inheritance, etc.). Nevertheless, in Section 4.3, we describe other approaches providing language support for design patterns.

4.1 Meta-Object-Protocol

One of the most flexible environments for object-oriented engineering is the CLOS environment with its meta-object-protocol [13]. We are convinced that filters can be implemented in this environment which provides many hooks to influence the behavior and semantics of objects. Our filter approach differs significantly, since filters provide a high level construct, which is tailored to monitor and to modify object interactions.

One example in [13] enhances CLOS with encapsulated methods capable of restricting the access of private variables to methods of their class. The system's method, used to apply methods, is enhanced with a sub-protocol which can add a set of function bindings to the method body's lexical environment. The filter would have been a shorter and higher level solution for this problem, because it does not require modifications or additions to the underlying systems behavior. Therefore it does not require knowledge about the systems structure, like how the system applies methods or how lexical definitions are bound to methods. Moreover, the filter solution can easier be scaled, since filters may be dynamically registered and unregistered.

4.2 Meta-Programming

From the abstraction point of view filters are closely related to the area of meta-programming, which was studied in the area of lisp-like languages (e.g. [2]) or in the area of logic languages, as sketched in this section.

The filter approach is a very general mechanism which can be used, besides language support for design patterns, in various other application areas. We see object-orientation and filters as an analogy to the interpretation layer introduced by meta-programs which are used to interpret existing programs in a new context with additional functionality [20, 21]. In [22] the abstraction introduced by layered interpreters is called interpretational abstraction. The basic idea of interpretational abstraction is to treat program instructions of one program (source program) as data of another program (a meta-program, a compiler or interpreter) that reasons about the instructions of the source program. During this reasoning process new functionality can be introduced into the source program by interpreting the goals of the source program in a new context. Instead of altering the application program (the knowledge representation), an additional interpretation layer can be introduced to change the behavior in certain situations. This way interpreters can be used as a programming device. The inefficiency of the reasoning process can be eliminated by techniques like partial evaluation [8] or interpreter directed compilation [21].

The filter approach is an introduction of meta-programming ideas into object-orientation. Even if the filter never accesses the real program (which a filter in XOTcl could do through the provided introspection mechanisms), it has full and unlimited access to the most important thing in object-oriented runtime structures: the messages. A filter handles messages of objects as data which can be processed in arbitrary ways (modified, redirected, handled). The filters are able to reinterpret messages freely, the filter methods are "interpreters" for messages and can influence all object communication.

In general the application domain of filters is very wide. For example assertions and meta-data as presented in [24] could have been implemented using filters. The only argument against this was, that the implementation in C is much faster than implementation using filters, since in the current implementation they reduce execution speed. However, it would be interesting to investigate, to what degree compilation methods like these described above could eliminate the overhead.

4.3 Other Approaches for Supporting Design Patterns

As stated above, Soukup [30] has identified problems in the implementation of popular design patterns [10] and has shown that some patterns could be implemented as classes.

The LayOM-approach [3] is the most similar to the filter approach. It offers an explicit representation of design patterns using an extended object-oriented language. The approach is centered on message exchanges as well and puts layers around the objects which handle the incoming messages. Every layer offers an interface for the programmer to determine the behavior of the layer through a set of operators which are (statically) given by the layer definition. LayOM is a compiled language with a static class concept and can be translated into C++. The model is statically extensible with new layers.

The filter approach differs from LayOM since it can represent design patterns as normal classes and needs no new constructs, only regular methods. Therefore, the filter approach is closer to the

object-oriented paradigm. Furthermore, the filters can be dynamically reconfigured (added, removed, etc.) and are able to exploit introspection provided by the underlying language.

The FLO-language [7] introduces a new component "connector" that is placed between interacting objects. The connectors are controlled through a set of interaction rules that are realized by operators (not normal methods). This connector-approach also concentrates on the messages of the objects but introduces the connectors as new entities. FLO is open for change (using a meta-object-protocol) and because the connectors are represented as objects it is close to the object-oriented paradigm.

The introduced operators are not object-oriented by nature and, therefore, less intuitive in an object-oriented system than method invocation. The approach of FLO involves a more complicated design, because in addition to the design patterns, connector objects have to be defined. The filter approach can avoid this problem by the automatic registration of filters on the involved classes.

Both mentioned approaches do not seem to offer the same ease as the filter in specializing an abstract pattern (like in [10]) to a concrete, more domain specific pattern (in the sense of [30]). Where the filters can simply use inheritance both approaches need the definition of a new domain specific layer or connector.

Hedin [12] presents an approach based on an attribute grammar in a special comment marking the pattern in the source code. This addresses the problem of traceability. The comments assign roles to the classes, which constrain them by rules like *"A DECORATOR must be a subclass of COMPONENT"*. The system can test automatically (in the source code) if the realized pattern satisfies the given and derived constraint rules.

This approach is not based on message exchanges (and is, therefore, rather simplistic), but it may be applied in any object-oriented language. It is only descriptive and not constructive (and, therefore, not reusable); each pattern must be commented again if it is applied to new application. The ability to assign constraints to patterns is interesting, especially because XOTcl provides similar abilities as well. The assertions can constrain classes (and objects) formally and informally. Both the consistency of the pattern class and its instances can be checked at run-time.

5 Conclusion

The intention of this paper is to show that object-oriented scripting languages and the management of complexity are not contradictory and that it is possible to handle complexity with a different set of advantages and tradeoffs than in "systems programming languages". Scripting is based upon several principles of programming, like using dynamic typing, flexible glueing of preexisting components, using component frameworks etc., that can lead towards a higher productivity and software reuse. We have introduced a new language construct, the filter, that offers a powerful means for the management of complex systems in a dynamic and introspective fashion. It would have been substantially more difficult to implement dynamic and introspective filters in a systems programming language. We believe that both scripting and object-orientation offer extremely useful concepts for a certain set of applications and that our approach is a useful and natural way to combine them properly.

XOTcl is available for evaluation from: `http://nestroy.wi-inf.uni-essen.de/xotcl/`

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa: *Abstracting Object Interactions Using Composition Filters*, ECOOP '93, 1993.

[2] H. Abelson, G.J. Sussman, J. Sussman: *Structure and Interpretation of Computer Programs*, MIT Press 1996.

[3] J. Bosch: *Design Patterns as Language Constructs*, http://bilbo.ide.hk-r.se:8080/~bosch/, 1996.

[4] J. Bosch: *Design Patterns and Frameworks: On the Issue of Language Support*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[5] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon: *Common Lisp Object System. In: Common Lisp the Language, 2nd Edition*, http://info.cs.pub.ro/onl/lisp/clm/node260.html, 1989.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture – A System of Patterns*, J. Wiley and Sons Ltd. 1996.

[7] S. Ducasse: *Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[8] A.P. Ershov: *On the Essence of Compilation*, in: E. Neuhold (ed.), IFIP Working Conference on Formal Descriptions of Programming Concepts, North-Holland, New York 1978.

[9] J.L. Fontain: *Simple Tcl Only Object Oriented Programming*, http://www.mygale.org/04/jfontain/, 1998.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994.

[11] L. Hatton: *Does OO Sync with How We Think?*, in: IEEE Software, Vol. 15 (3), 1998.

[12] G. Hedin: *Language Support for Design Patterns using Attribute Extension*, Workshop on Language Support for Design Pattern Frameworks at ECOOP'97, Jyväskylä 1997.

[13] G. Kiczales, J. des Rivieres, D.G. Bobrow: *The Art of the Metaobject Protocol*, MIT Press 1991.

[14] G.E. Krasner, S.T. Pope: *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object Oriented Programming, Vol. 1 (3), pp. 26-49, 1988.

[15] M.J. McLennan: *The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More*, Proceedings of the Tcl/Tk Workshop '95, Toronto 1995.

[16] IONA Technologies Ltd.: *The Orbix Architecture*, August 1993.

[17] H. Lieberman: *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings OOPSLA '86, 1986.

[18] B. Meyer: *Object-Oriented Software Construction - Second Edition*, Prentice Hall 1997.

[19] B. Meyer: *Building bug-free O-O software: An introduction to Design by Contract*, http://eiffel.com/doc/manuals/technology/contract/index.html, 1998.

[20] G. Neumann: *Meta-Programmierung und Prolog*, Addison-Wesley 1988.

[21] G. Neumann: *A simple Transformation from Prolog-written Metalevel Interpreters into Compilers and its Implementation*, Lecture Notes in Artificial Intelligence 592, Springer, Berlin 1992.

[22] G. Neumann: *Interpretational Abstraction*, in: Computers and Mathematics with Applications, Pergamon Press, Vol. 21, No. 8, 1991.

[23] G. Neumann, S. Nusser: *Wafe - An X Toolkit Based Frontend for Application Programs in Various Programming Languages*, USENIX Winter 1993 Technical Conference, San Diego, California, January 1993.

[24] G. Neumann, U.Zdun: XOTcl, *an Object-Oriented Scripting Language, submitted*, 1998.

[25] J. Ousterhout: *Tcl: An embeddable Command Language*, Proceedings of the 1990 Winter USENIX Conference, 1990.

[26] J. Ousterhout: *Scripting: Higher Level Programming for the 21st Century*, in: IEEE Computer, Vol. 31, No. 3, March 1998.

[27] I. Piumarta: *SSP Chains - from mobile objects to mobile computing (Position Paper)*, ECOOP Workshop on Mobility, , Aarhus 1995.

[28] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley 1995.

[29] J. Smith, D. Smith: *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems, 2:2, June 1977.

[30] J. Soukup: *Implementing Patterns*, in: J.O. Coplien, D.C. Schmidt (Eds.), *Pattern Languages of Program Design*, Addison-Wesley 1995, pp 395-412, 1995.

[31] P. Wegner: *Learning the Language*, in: Byte, Vol. 14, No. 3, pp. 245-253, March 1989.

[32] D. Wetherall, C.J. Lindblad: *Extending Tcl for Dynamic Object-Oriented Programming*, Proceedings of the Tcl/Tk Workshop '95, Toronto, July 1995.

[33] U. Zdun: *Entwicklung und Implementierung von Ansätzen, wie Entwurfsmustern, Namenräumen und Zusicherungen, zur Entwicklung von komplexen Systemen in einer objektorientierten Skriptsprache*, Diplomarbeit (diploma thesis), Universität GH Essen, August 1998.

Performance Patterns: Automated Scenario Based ORB Performance Evaluation*

S. Nimmagadda, C. Liyanaarachchi, A. Gopinath†, D. Niehaus, A. Kaushal†
Information and Telecommunication Technology Center
Electrical Engineering and Computer Science Department
University of Kansas

†Sprint Corporation

Abstract

The performance of CORBA *(Common Object Request Broker Architecture)* objects is greatly influenced by the *application context* and by the performance of the ORB *endsystem*, which consists of the middleware, the operating system and the underlying network. Application developers need to evaluate how candidate application object architectures will perform within heterogenous computing environments, but a lack of standard and user extendable performance benchmark suites exercising all aspects of the ORB endsystem under realistic application scenarios makes this difficult. This paper introduces the *Performance Pattern Language* and the *Performance Measurement Object* which address these problems by providing an automated script based framework within which extensive ORB endsystem performance benchmarks may be efficiently described and automatically executed.

1 Introduction

The Common Object Request Broker Architecture (CORBA)[15] is emerging as an important open standard for distributed-object computing, especially in heterogenous computing environments combining multiple platforms, networks, applications, and legacy systems[27]. Although the CORBA specifications define the features of a compliant ORB, they do not specify how the standards are to be implemented. As a result, the performance of a given application supported by ORBs from different vendors can differ greatly, as can the performance of different applications supported by the same ORB.

A number of efforts have been made to measure the performance of ORBs, often comparing with performance of other ORBs [23]. These efforts generally measure only specific aspects of ORB performance in isolation. While performance of specific ORB functions is important, it is also important to realize that superior results in a few simple tests *does not ensure* that the aggregate performance of ORB A is better than ORB B for a particular application object architecture. The performance of ORB based applications implemented as a set of objects is greatly influenced by the application context and by the architecture and performance of the ORB endsystem. The endsystem consists of the ORB middleware, the operating system and the underlying network. An application's performance is determined by how well these components cooperate to meet the particular needs of the application.

Current benchmark suites and methods tend to concentrate on a specific part of the endsystem. Operating system benchmarks concentrate on component operating system operations, but may say comparatively little about how well the operating system will support ORB middleware. ORB benchmarks concentrate on component operations of the middleware, but are less effective at pinpointing problems at the application, operating system, and network layers. Developers considering non-trivial ORB based applications need the ability to evaluate, in some detail, how well a given ORB and endsystem combination can support candidate application object architectures. They need this information before implementing a significant portion of the entire application. Such developers should begin with a set of standard performance benchmark suites exercising various aspects of the ORB endsystem under realistic application scenarios, but they also require the ability to create test scenarios which specifically model their candidate application architectures and behavior in the endsystem context.

Current benchmarking methods and test suites do not adequately solve the real problem developers face be-

*This work was supported in part by grants from Sprint Corporation.

cause current methods concentrate on only a part of the application and endsystem in isolation and thus do not enable the developer to consider how implementation decisions at various levels interact. An effective and efficient tool set supporting an integrated performance evaluation methodology should support ORB, endsystem, and application oriented tests, should be automated, and should make it easy for the user to extend and modify the set of tests performed. Only such an integrated tool set and benchmark test suite supporting realistic application scenarios and capable of collecting information from all layers of the endsystem can enable developers to effectively evaluate candidate application object architectures *before* implementation.

This paper describes how a combination of tools developed at the University of Kansas (KU) can address this challenge. This integrated tool set represents a significant advance in support for performance evaluation of ORB based applications because it increases the range and complexity of tests that a benchmark suite can contain, it extends the types of performance information which can be gathered during an individual test, and its support for automated test execution significantly extends the number of tests that a practical benchmark suite can contain. The NetSpec tool provides a control framework for script driven automation of distributed performance tests. The Data Stream Kernel Interface (DSKI) provides the ability to gather time stamped events and a variety of other performance data from the operating system as part of a NetSpec experiment. The Performance Measurement Object (PMO) provides the ability to conduct NetSpec based experiments involving CORBA objects, and the Performance Pattern Language (PPL) provides a higher level language for describing NetSpec based experiments involving sets of CORBA objects more succinctly.

NetSpec has been used by a number of research projects at the University of Kansas (KU) and elsewhere. It provides the automation and script based framework supporting experiments including a wide range of conditions, component behaviors, and data collection [12, 16]. NetSpec is designed to be extended and modified by the user through the implementation of *daemons*. Test daemons support basic network performance tests and supply background traffic in other NetSpec based experiments. Measurement daemons gather information during an experiment but contribute no traffic or behavior beyond that required to gather data.

The DSKI is a pseudo-device driver which enables a NetSpec experiment, through the DSKI measurement daemon, to specify and collect the set of operating system level events of interest which occur during the experiment [1]. The PMO is a NetSpec test daemon designed to support CORBA based performance experiments. A NetSpec PMO script can specify the creation of CORBA objects, their execution time behavior, and the relations that hold among the objects. Using existing traffic related NetSpec test daemons, the DSKI, and PMO, a user can write a script specify a set of interacting objects, a set of network background traffic providing a context within which the objects exist, and gather operating system level information about network and operating system level events affecting performance.

A practical drawback of the NetSpec PMO support is that the language is defined at a low level of detail, and PMO scripts for scenarios with many objects are thus long and repetitive. The PPL addresses this by defining a higher level language for more compactly describing application level object interaction scenarios, which abstract the performance aspects of commonly used implementation strategies. We have called these scenarios *performance patterns* to draw a direct analogy to design patterns which the definitive book *Design Patterns* defines on page 3 as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"[6].

A performance pattern is a set of objects exhibiting a set of behaviors, relationships, and interactions typical of an application architecture or class of application architectures. This pattern can be customized through parameter specification or user extension to match the intended application behaviors and architecture as closely as required. The PPL compiler emits NetSpec PMO scripts implementing the specified performance pattern.

It is important to realize that the PPL approach is quite general and is not ORB or even CORBA specific. The PPL could easily be used to create object based performance scenarios given support from a NetSpec daemon of the correct type. The PMO is CORBA specific, but it would be straightforward to implement and analogous NetSpec daemon for DCE or DCOM based performance evaluation. The PMO is not ORB specific and has been ported with minimal effort to four ORBs: The ACE ORB (TAO)[20], OmniORB[23], ExperSoft's CORBAplus[5], and ILU[11]. We currently focus on TAO, OmniORB, and CORBAPlus for project specific reasons. The range of experiments which can be supported is a function, in part, of the set of possible object behaviors supported by the PMO. PMO behaviors are implemented by routines linked into the PMO, and it has been designed to make adding new behaviors simple, thus supporting user extension.

The rest of the paper first discusses related work in Section 2, and then describes the implementation of the PMO and PPL in Section 3. Section 4 presents examples of PMO and PPL use, while Section 5 presents our conclusions and discusses future work.

2 Related Work

A number of efforts have been made to measure the performance of ORBs, often comparing with performance of other ORBs [23]. Earlier studies on the performance of CORBA objects focussed mainly on identifying the performance constraints of an Object Request Broker (ORB) alone. Schmidt analyzed the performance of Orbix and VisiBroker over high speed ATM networks and pointed out key sources of overhead in middleware ORBs [7, 8]. This paper complements Schmidt's work by demonstrating an integrated and automated approach which is capable of simultaneously measuring the influence of the ORB, the operating system and the underlying network on the performance of CORBA objects.

Studies have also been conducted on IDL compiler optimizations that can improve overall performance of the ORB. One such effort is the Flick project[4]. It is claimed that Flick-generated stubs marshal data between 2 and 17 times faster than stubs produced by traditional IDL compilers, resulting in an increased end-to-end throughput by factors between 1.2 and 3.7. While clearly addressing an important topic, the Flick work also clearly concentrates on one specific facet of endsystem performance. Our work complements such efforts by providing a platform within which the effect of such efforts on endsystem performance can be evaluated.

TAO is the ACE ORB being developed at Washington University [17]. This project focuses on: (1) identifying the enhancements required to standard ORB specifications that will enable applications to specify their Quality of Service (QoS) requirements to ORBs, (2) determining features required to build real-time ORBs, (3) integrating the strategies for I/O subsystem architectures and optimizations with ORB middleware, and (4) to capture and document key design patterns necessary to develop, maintain and extend real-time ORB middleware. The work described in this paper compliments these goals by providing a way to capture, document, and evaluate performance aspects of ORB based design patterns.

In addition to providing a real-time ORB, TAO is an integrated ORB endsystem architecture that consists of a high-performance I/O subsystem and an ATM port inter-connect controller (APIC). They have developed a wide range of performance tests which include throughput tests[7], latency tests[8] and demultiplexing tests[8]. They have used these performance tests to test TAO [17] and other CORBA2.0 compliant ORBs. Their tests formed a basis for several of the basic tests in the automated framework described in this paper.

A commercially available CORBA test suite is the VSORB from X/Open [28]. VSORB is implemented under the TETware test harness, a version of the Test Environment Toolkit (TET), a widely used framework for implementing test suites [24]. It is designed for two primary uses: (1) testing ORB implementations for CORBA conformance and interoperability under formal processes and procedures, and (2) CORBA compliance testing by ORB implementors during product development and quality assurance. This work differs from ours in that it concentrates on compliance rather than performance, but clearly shares the goal of creating a general framework for large scale evaluation tests.

The Manufacturing Engineering Laboratory at the National Institute of Standards & Technology(NIST) takes a different approach towards the benchmarking of CORBA in their current work on the Manufacturer's CORBA Interface Testing Toolkit(MCITT) [13]. They use a emulator-based approach in which the actual servers are replaced by test servers and the person doing the testing only needs to specify the behaviors that are important for the specific scenario being examined. The approach provides an extremely simplified procedural language, the Interface Testing Language, for specifying and testing the behavior of CORBA clients and servers. This work is similar to ours with respect to its abstraction of the object behavior, but it does not explicitly integrate endsystem evaluation, concentrating only on the application and ORB middleware.

The Distributed Systems Research Group at Charles University, Czech Republic, have done a comparison of three ORBs based on a set of criteria including dispatching ability of the ORB, throughput provided for the invocation of different data types, scalability, and performance implications of different threading architectures [2, 19]. The criteria address different aspects of the ORB functionality and the influence of each criterion has been discussed with respect to specific ORB usage scenarios. They have also developed a suite of benchmarking applications for measurement and analysis of ORB middleware performance. This is a strong effort, but the drawback to this approach, in our view, is that it is restricted to evaluating ORB level performance and specific predefined application scenarios. This is significant because application behaviors will vary and their method does not appear, by our understanding, to be designed to support user specified test scenarios.

Performance evaluation is an important topic in many areas of computer system design and implementation, and significant related work exists which does not consider ORB performance. Data bases provide some of the best developed examples of benchmarks addressing application scenarios. It is interesting to observe that both data bases and ORBs support applications by assuming the role of middleware. As such, performance evaluation of data bases is most meaningful and useful to potential users when it considers application scenarios.

The Wisconsin Benchmark is an early effort to systematically measure and compare the performance of relational database systems with database machines[3]. The benchmark is a single-user and single-factor experiment using a synthetic database and a controlled workload. It measures the query optimization performance of database systems with 32 query types to exercise the components of the proposed systems. This is similar to our effort in that it abstracts the application scenario and considers a range of system functions. Our work differs, however, in that we also provide for placing the set of ORB based objects in an endsystem context including background load and traffic.

The ANSI SQL Standard Scalable and Portable Benchmark (AS3AP) models complex and mixed workloads, including single-user and multi-user tests, as well as operational and functional tests [25]. There are 39 single-user queries consisting of utilities, selection, join, projection, aggregate, integrity, and bulk updates. The four multi-user modules include a concurrent random read test and a pure information retrieval (IR) test[26]. The concurrent random write test is used to evaluate the number of concurrent users the system can handle updating the same relation. The mixed IR test and the mixed OLTP test are to measure the effects of the cross-section queries on the system with concurrent random reads or concurrent random writes. This effort has a stronger similarity to ours in that it considers a wider range of activity as well as multiple users. It does not, to our knowledge, provide support for users to specify application based test scenarios.

3 Implementation

We have implemented *an integrated tool-based approach* for performance measurement of ORB endsystem performance. The single most important aspect of our system is that it measures performance within the target environment, rather than relying on published data that may be inaccurate, or which accurately describes aspects of performance under a different environment. The main features of this approach are:

1. A script based approach for conducting performance tests which promises better expressiveness of experiments.
2. The ability to study the performance of CORBA objects in the context of different operating system loads and network traffic.
3. The ability to study the influence of different components of the CORBA endsystem including the middleware, the operating system, and the network on the performance of CORBA objects.
4. The ability to measure the performance of objects in heterogeneous distributed systems from a single point of control.
5. The flexibility and scalability to specify a wide range of distributed tests and behavior patterns. This includes scalability in time, number of objects, and number of hosts supporting the pattern.
6. The ability to measure latencies, throughput and missed deadlines among a wide range of performance metrics.
7. An automated highly scalable framework for performance measurement. This is a crucial feature because it enables practical us of much larger benchmarking suites than non-automated approaches.

The performance metrics which best predict application performance depend, in part, on the properties of the application. This is one reason why a pattern based and automated framework is required. The pattern orientation enables the user to describe scenarios with a rich and varied set of behaviors and requirements, closely matching the proposed application architecture. Automation enables testing on a large scale, permitting the user to test a wide range of parameters under a wide range of conditions, which permits the user to *avoid* making many potentially unjustified assumptions about what aspects of the application, ORB, and endsystem are important in determining performance.

The metrics which will be crucial for important classes of applications include: throughput, latency, scalability, reliability and memory use. The system parameters which can affect application performance with respect to these metrics include: multi-threading, marshalling and demarshalling overhead, demultiplexing and dispatching overhead, operating system scheduling, integration of I/O and scheduling, and network latency. Our approach currently enables us to examine the influence of many of these aspects of the system on performance, and further development will enable us to handle all of them.

Figure 1 shows our integrated benchmarking framework supporting performance evaluation tests. The experiment description expressed in the PPL script is parsed by the PPL compiler which emits a PMO NetSpec script implementing the specified experiment. The NetSpec parser processes the PMO based script and instructs the NetSpec controller daemon to create the specified sets of daemons on each host used by the distributed experiment. Note that Figure 1 illustrates a generic set of daemons, rather than those supporting a specific test. The PMO daemon interfaces the

CORBA based objects on that host to the NetSpec controlling daemon. An additional PMO object is sometimes used, and communicates with the PMO daemon, because CORBA objects can be created dynamically. Note that the line between the PMO objects represents their CORBA based interaction, which is the focus of the experiment. The DSKI measurement daemon, if present, is used to gather performance data from the operating system. It is a generic daemon and is not CORBA based. The traffic daemon is also not CORBA based, but is used to create a context of system load and background traffic within which the CORBA objects exist.

Our approach integrates several existing tools and adds significant new abilities specifically to support CORBA. The tools integrated under this framework are *NetSpec*[12, 16], the *Data Stream Kernel Interface* (DSKI)[1], the *Performance Measurement Object* (PMO)[10, 9], and the *Performance Pattern Language* (PPL). The rest of this section discussed each component in greater detail.

3.1 NetSpec

NetSpec has been used by a number of research projects at the University of Kansas (KU) and elsewhere. It provides the automation and script based framework supporting experiments including a wide range of conditions, component behaviors, and data collection [12, 16]. NetSpec is designed to be extended and modified by the user through the implementation of daemons supporting specific component roles in experiments. Test daemons are used as active components, traffic sources and sinks, while measurement daemons are passive with respect to the experiment since they only collect measurements. Existing NetSpec daemons support network level performance tests with many simultaneous connections and traffic load profiles, as well as data collection from both hosts and network nodes using measurement daemons. A wide range of NetSpec daemons exist, providing a range of behaviors and functions, including: TCP/UDP traffic load, ATM signaling load, SNMP data collection, and DSKI data collection from the operating system.

3.2 Data Stream Kernel Interface

The DSKI is a pseudo-device driver which enables a NetSpec experiment, through the DSKI measurement daemon, to specify and collect a series of time-stamped operating system level events of interest which occur during the experiment [1]. This is particularly useful when considering interactions among the application, middleware, and operating system levels of the endsystem. The primary target platform for the DSKI is Linux, but we have also ported it to DEC UNIX, and as a pseudo-device driver it can be ported relatively easily to any version of UNIX. The DSKI supports a range of data collection options with differing in level of detail and overhead. One particularly powerful feature is the ability to associate and arbitrary *tag* with an event. For example, when the tag is a packet ID or buffer address this enables post processing to track the progress of specific messages through the protocol stack. In the CORBA context, post processing of the event stream shows the amount of time spent by messages in different portions of the operating system when making object request calls to and from the ORB. We are currently working to create a similar ability to create, configure, and process streams of events from the CORBA and application levels.

3.3 Performance Measurement Object

The PMO is a NetSpec test daemon designed to support CORBA based performance experiments. The PMO enables a NetSpec script to specify the creation of CORBA objects, their execution time behavior, and the relations that hold among the objects. The PMO control layer parses the instructions from the NetSpec controller specifying its role within an experiment. These objects can exhibit a variety of behaviors, and are capable of exchanging a wide range of CORBA data with each other.

The PMO provides all the basic abilities required to conduct CORBA based evaluation experiments, but experience has shown that it is not always the best way. The reason for this is that NetSpec's method of ensuring user extendibility and portability also ensures that NetSpec scripts are very long. The best analogy is to consider the NetSpec scripting language an architecture independent assembly language. It is thus possible to describe any desired experiment, but sometimes tedious. The PMO level is appropriate for describing basic CORBA component tests, but can be unwieldy when used for application level object interaction scenarios. The PPL addresses this problem, and is discussed in Section 3.4

The PMO NetSpec script language describes an experiment in terms of sets of daemons. Each daemon specification provides a complete list of parameter-value pairs describing that daemons role in the experiment. Groups of daemons are created and executed by the NetSpec controller either in *serial* or in *parallel*. These simple constructs make it possible to describe a wide range of sophisticated application level behaviors. Additional constructs make it possible to have sets of distributed subordinate controller daemons for large scale distributed experiments. The details of the NetSpec syntax are described elsewhere [12, 16], but the examples

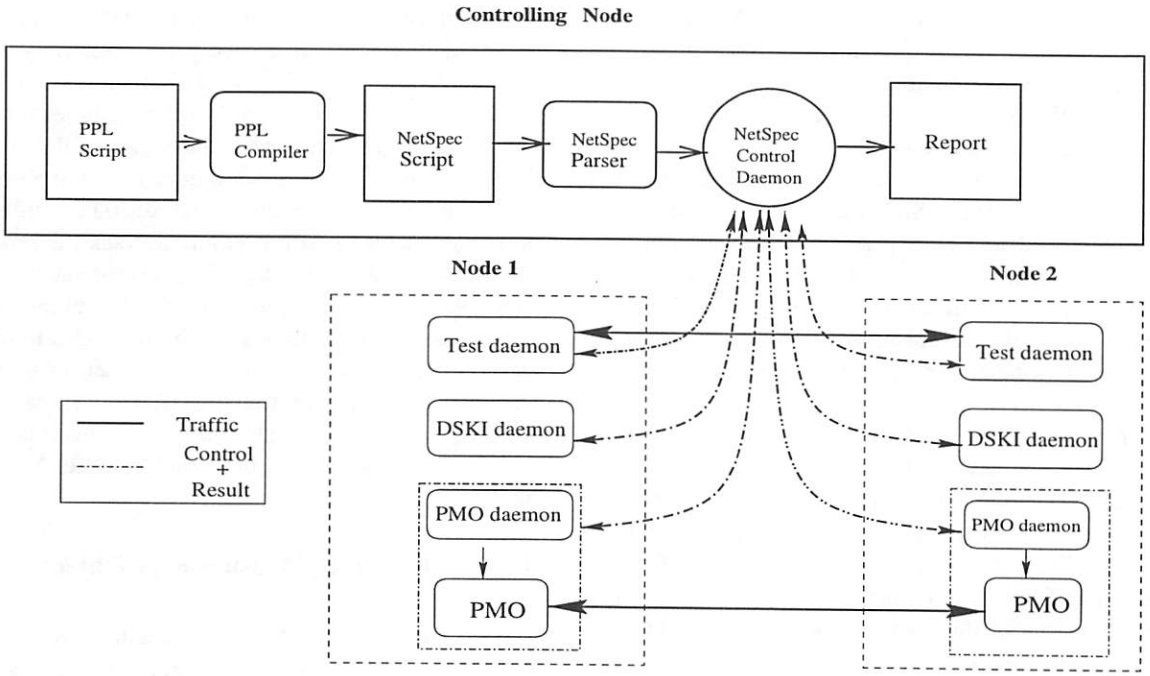


Figure 1. The Integrated Benchmarking Framework

described in Section 4 should provide a clear idea of how the system works.

3.4 Performance Pattern Language

The PPL was designed as a higher level language for describing such application level object interaction scenarios[14, 6] in terms of *performance patterns*. Within each pattern, the user describes objects, object behaviors, test types and relations among the objects that influence the performance of the pattern as a whole. For convenience, the PPL also permits the user to define parameter blocks describing aspects of object behavior which are referenced by object definitions using the same set of parameters. After the patterns associated with the experiment are specified, the *schedule* for their execution is given. Currently the only schedule supported is a simple sequential execution of one pattern at a time. However, we are working on extending this to permit flexible pattern composition and dynamic time dependent behavior to better support application scenario based testing. The correspondence between PPL constructs and the PMO level scripts produced by the PPL compiler is illustrated by the examples in Section 4.

The combination of the PMO and PPL provides a powerful and efficient way for developers to describe and conduct a wide range of application scenario based performance evaluation experiments for CORBA systems. The method is applicable to any ORB and has been ported with minimal effort to four ORBs: The ACE ORB (TAO)[20], OmniORB2[23], ExperSoft's CORBAplus[5], ILU[11]. The range of experiments which can be supported is a function, in part, of the set of possible object behaviors supported by the PMO. PMO behaviors are implemented by routines linked into the PMO, and it has been designed to make adding new behaviors simple.

The scalability of our method is important in two ways. First, script driven automation of the experiments makes it fairly easy to describe tests at a scale representative of the final application. Second, the script driven automation makes it possible to conduct an acceptably large and comprehensive set of tests in an acceptably short period of time. For example, sets of tests producing graphs discussed in Section 4 are fully automated and execute in periods ranging from a few seconds to almost an hour. Scalability is important because the number of properties of an ORB which can significantly affect performance of a particular application is large, requiring a large test suite for adequate evaluation.

4 Evaluation

This section illustrates current capabilities as well as the potential of our automated script driven and application scenario based performance evaluation methods

and tools. The examples show how the tests used in current benchmarks are supported by the framework, and how these can be used as components of more sophisticated scenario based performance patterns. This section presents results of two types of tests under two patterns to illustrate our methods. Section 4.1 presents results under the simple client-server pattern and behaviors for the *cubit* and *throughput* test types. Section 4.2 presents the results under the *proxy* pattern for the same behaviors and test types. Section 4.3 demonstrates the use of the DSKI to reveal the components of the system support overhead for the client-server pattern using a simple request-response behavior. We also demonstrate the portability of our method by presenting results for both Linux and Solaris. Table 1 presents the Linux testing environment for the *cubit* and *throughput* behavior tests, while Table 2 presents that for Solaris. Note that the sending machine is slightly slower than the receiving machine. We originally used identical machines, but a machine failure forced us to use a different receiving machine for tests presented here.

Name of ORB	omniORB2, TAO
Language Mapping	C++
Operating System	Redhat Linux 5.1 kernel 2.1.126
CPU info	Pentium Pro 200 MHz 128 MB RAM
Compiler info	egcs-2.90.27 (egcs-1.0.2 release) no optimizations
Thread package	Linux-Pthreads 0.7
Type of invocation	static
Measurement method	getrusage
Network Info.	ATM

Table 1. Operating Environment Used for the Tests on Linux Platform

Significant further development of our approach is desirable, and is proceeding, but the current capabilities of the tools generally meet and modestly exceed some aspects of current practice. It is important to note that the framework is explicitly designed for user extension precisely because no single developer or authority can know every significant aspect of ORB evaluation. Accumulation of the sum of the CORBA community's collective wisdom concerning ORB evaluation would significantly advance the state of the art. The script based automated approach described here is designed to support such a collective effort.

Parameter	Description
Name of ORB	omniORB2, TAO, CORBAplus
Language Mapping	C++
Operating System	Solaris 2.6
CPU info	Ultra Sparc-II 296 MHz (S) Ultra Sparc-IIi 350 MHz (R) 128 MB RAM
Compiler info	SUN C++ 4.2 no optimizations enabled
Type of invocation	static
Measurement method	getrusage
Network Info.	ATM

Table 2. Operating Environment Used for the Tests on Solaris Platform

4.1 Simple Client-Server Pattern

This example illustrates the basic elements of the PPL and PMO in the context of a simple client-server pattern, which reflects current conventional benchmarking practice. Listed below is the PPL script corresponding to the scenario of Figure 2. The client and server in this case are Sender and Receiver respectively. The information regarding the parameters required for the testing between these two CORBA objects is provided in the *object blocks* of the PPL script and the kind of relation between the objects is specified in the *relation block* of the PPL script. Execution of the pattern is specified by the one line schedule.

The PPL compiler takes the script as input, analyzes the object definitions and relations, and generates the NetSpec PMO script shown in Figure 3. The first thing to note is that the PMO script has two major sections, one defining the client as a *corba* daemon running on the machine *marcus*, and the server as a *corba* daemon running on the machine *zeno*. The other major point is that the parameter block is specified explicitly for each daemon. The main point is that the PMO script defines each object separately and that the relations among them are more difficult to discern in the PMO language.

Figure 4 shows the performance of the Client-Server pattern supporting the *cubit* test type for OmniORB and TAO on a Linux platform, while Figure 5 shows the results for OmniORB, TAO and CORBAplus on a Solaris platform. The CORBAPlus ORB is not currently available for Linux, but should be soon. The flexibility of the script driven approach is demonstrated by the observation that the TAO based tests were repeated for the OmniORB by replacing *orb_name = TAO* with *orb_name = OmniORB* in the PPL script. The *cubit* test emphasizes basic communication performance because it involves


```
pattern CUBIT-TESTS {
  param_block param1 {
    test_type = cubit; orb_name  = TAO;
    minsize   = 512;   maxsize   = 8192;
    predelay  = 5;     postdelay = 5;
    duration = 10;     multiples = 2;
    protocol = iiop;         qos = normal;
    criteria  = latency;
  }

  object Sender  {
    machine_name = marcus; interface = eth;
    behavior = client;        param  = param1;
    numsamples   = 250;
  }

  object Receiver {
    machine_name = zeno;   interface = eth;
    behavior     = server; param     = param1;
    numsamples   = 250;    port_num  = 22222;
  }

 relations {
    (TAO-sender,TAO-receiver);
  }
}

/* Execution Schedule */

CUBIT-TESTS;
```

Figure 2. Simple Client-Server Pattern

```
cluster {
  corba  marcus {
    NameOfORB  = TAO;
    TypeOftest = cubit;
    TestParams = (
      numsamples = 250,  minsize    = 512,
      maxsize    = 8192, multiples  = 2,
      predelay   = 5,    postdelay  = 5,
      duration   = 10 );
    protocol = iiop;
    objname  = Sender;
    role     = client;
    relations = server{Receiver};
    criteria = latency;
    qos      = normal;
    own      = marcus (interface = eth);
  }

  corba  zeno {
    NameOfORB  = TAO;
    TypeOftest = cubit;
    TestParams = (
      numsamples = 250,  minsize    = 512,
      maxsize    = 8192, multiples  = 2,
      predelay   = 5,    postdelay  = 5,
      duration   = 10 );
    protocol = iiop;
    objname  = Receiver;
    role     = server;
    relations = client{Sender};
    criteria = latency;
    qos      = normal;
    own      = zeno (interface = eth, port = 22222);
  }
}
```

Figure 3. Corresponding Client-Server PMO NetSpec Script

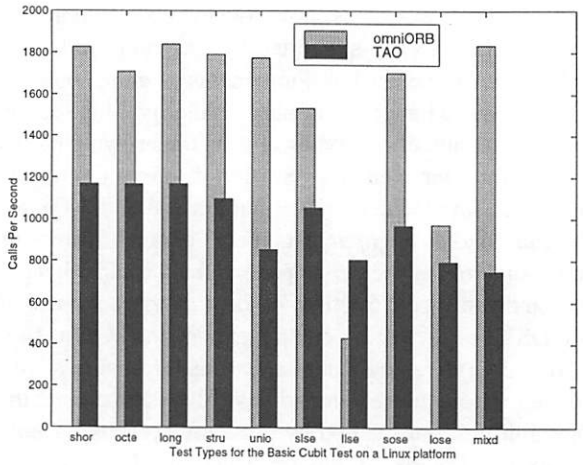


Figure 4. Client-Server Cubit for OmniORB and TAO on Linux

small packets, and a simple computation (cube a number) on the server side. The *cubit* behavior thus focuses on the time spent by each packet in the system layers and the middleware for a CORBA call invocation. The results shown are the average values for 250 invocations of the basic operation for each of several CORBA data types, and are presented in terms of calls per second.

There are several points of interest in these results. First, is the fact that even such a simple test reveals differences between ORB implementations, and between operating system platforms. The most striking difference is that while TAO performance is essentially constant on both Linux and Solaris, OmniORB performance on Solaris is roughly double that on Linux for many data types but not all. Another observation is that OmniORB generally outperforms the other ORBs, but that its performance for the "llse" (long long sequence) data type is substantially below that of TAO on Linux.

Determining why these observed behaviors occur will take further study, but this demonstrates the important point that our compact PPL script describes a test which can be run automatically in a matter of seconds, revealing significant differences in ORB behavior, and providing a convenient and efficient foundation for further experimentation. The flexibility of the PPL approach is further illustrated by changing the test type from *cubit* to *throughput* in the client-server pattern, producing the Linux throughput results for TAO illustrated in Figure 6 and Figure 7 for OmniORB. A data file, essentially CORBA "char" data type, ranging from 1 MB to 64 MB is sent using buffer sizes ranging from 512 bytes to 16 KB. This test shows that throughput for both ORBs is constant with the total amount of data, but that throughput is significantly affected by the

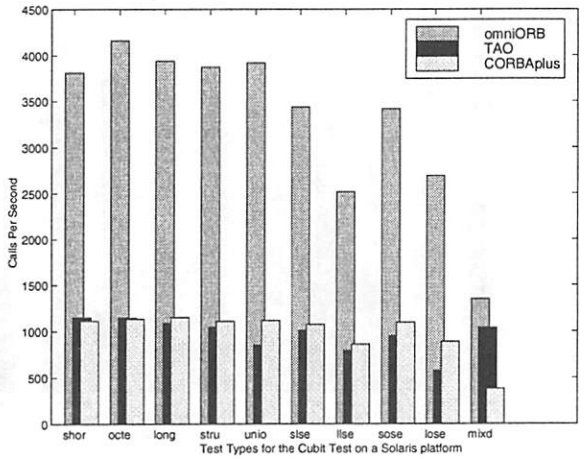


Figure 5. Client-Server Cubit for OmniORB, TAO and CORBAplus on Solaris

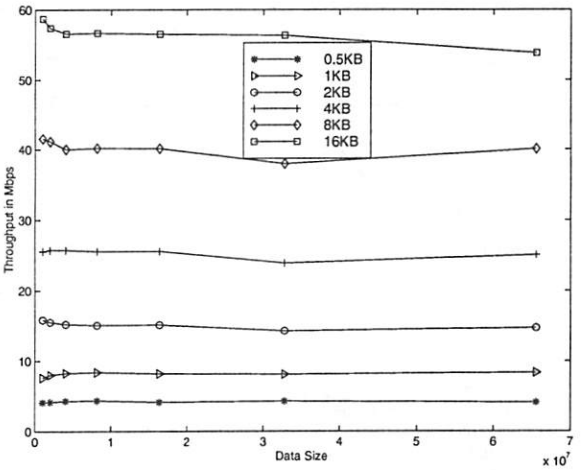


Figure 6. Client-Server Throughput for TAO on Linux

buffer size used for each data transfer session. The TAO throughput increased with buffer size, indicating that the packet transfer rate was limited, but not the packet size. The OmniORB throughput varied in a much less obvious way, and was significantly greater for 4KB buffers. Determining why OmniORB performance varies so haphazardly with buffer size would require gathering data from the operating system layer, as discussed in Section 4.3. The throughput tests for a single client-server pair ran under NetSpec control in an elapsed time of approximately 15 minutes.

4.2 Proxy Pattern

This section discusses a more complex CORBA application scenario, the Proxy Pattern [22], in which the proxy object acts as an interface between the CORBA clients and CORBA servers as shown in Figure 8 for three client and server objects, with the PPL script implementing this pattern for the *cubit* test type under OmniORB. The proxy pattern uses the basic client-server pattern as a component, extending it to a group of client-server pairs communicating through a proxy object. In this case we use three client server pairs under the proxy pattern which exhibit the client and server behaviors, respectively, while executing the *cubit* and *throughput* test types. The client contacts the corresponding server at run-time either by passing the object reference, or the server's name registered with the CORBA Naming Service, to the Proxy object which forwards the client request to the appropriate server. The data type used for the transfer of information between the clients and the proxy Object is CORBA "Any".

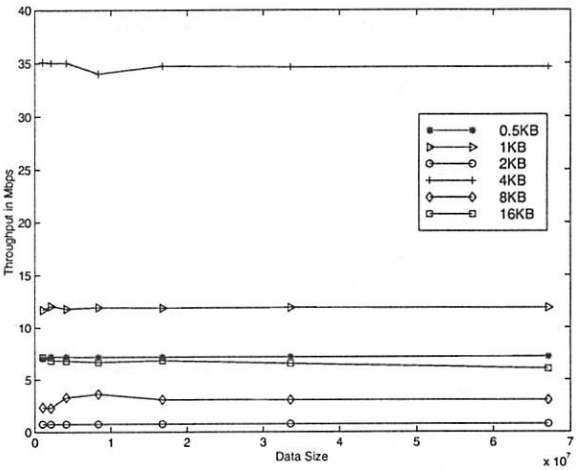
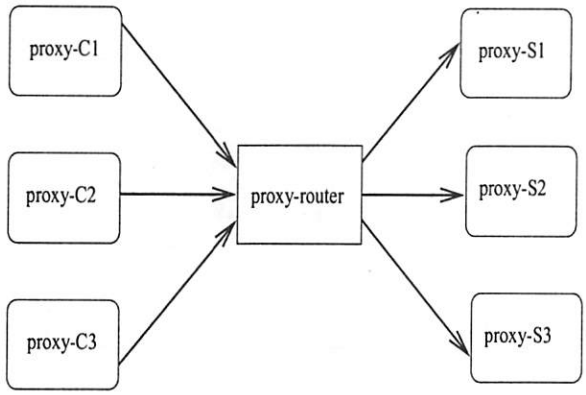


Figure 7. Client-Server Throughput for OmniORB on Linux




```
pattern Proxy  {
  param_block client-param {
    orb_name = omniORB2; test_type = cubit;
    numsamples = 250;    minsize   = 1;
    maxsize    = 1;      multiples = 1;
    predelay   = 3;      postdelay = 3;
    protocol = iiop;           qos = normal;
    criteria = latency;  interface = eth;
    machine_name = marcus;
  }

  param_block server-param {
    orb_name = omniORB2; test_type    = cubit;
    predelay = 3;        postdelay    = 4;
    interface = eth;     machine_name = zeno;
  }

  object proxy-C1 {
   behaviour = client; param = client-param;
  }
  object proxy-C2 {
   behaviour = client; param = client-param;
   predelay  = 5;      postdelay  = 3;
  }
  object proxy-C3 {
   behaviour = client; param = client-param;
   predelay  = 7;      postdelay  = 3;
  }
  object proxy-S1 {
   behaviour = server; param = server-param;
   port_num  = 10000;
  }
  object proxy-S2 {
    behaviour = server; param = server-param;
    port_num  = 20000;
  }
  object proxy-S3 {
    behaviour = server; param = server-param;
    port_num  = 30000;
  }
  object proxy-router {
    behaviour = proxy ; param = server-param;
    port_num  = 30003;
  }

  relations {
    (proxy-C1,proxy-router); (proxy-C2,proxy-router);
    (proxy-C3,proxy-router); (proxy-router,proxy-S1);
    (proxy-router,proxy-S2); (proxy-router,proxy-S3);
  }
}

/* Execution Schedule */
Proxy;
```

Figure 8. Proxy Pattern and PPL Script

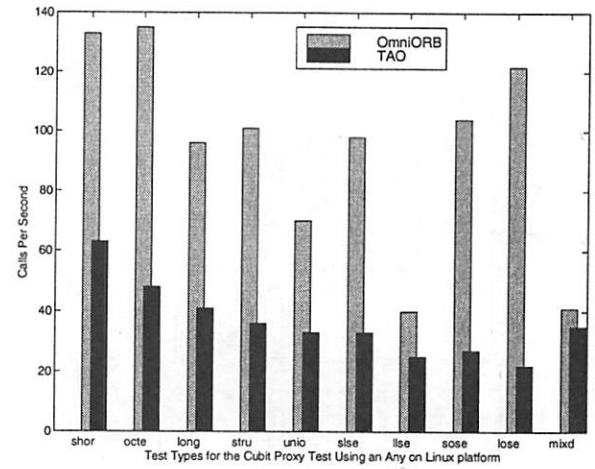


Figure 9. Proxy Cubit Results for OmniORB and TAO on Linux

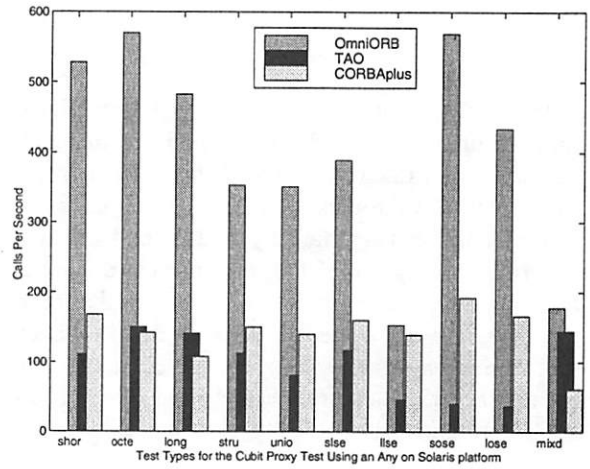


Figure 10. Proxy Cubit Results for OmniORB, TAO and CORBAplus on Solaris

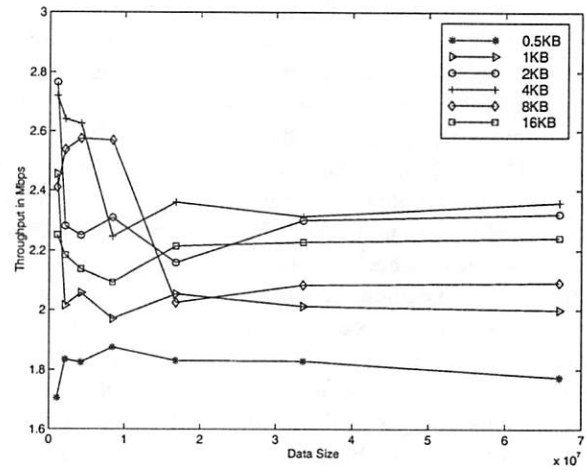


Figure 11. Proxy Throughput for OmniORB on Solaris

The proxy object can be used in two modes. In the first, the proxy only plays a role when establishing a connection between the client and server. In the other mode, the proxy actually routes the data between the objects, with a significant effect on performance. We present results for the second mode.

Figure 9 shows the performance of the omniORB and TAO client objects using the *cubit* test type under the proxy pattern on Linux, while Figure 10 shows the performance of clients under those ORBS as well as under CORBAplus on Solaris. The number of calls per second shown in Figure 9 are the average of the numbers of the three clients in both OmniORB and TAO. There was some non-trivial variance among clients for some tests and some ORBs, which would be another interesting point for further investigation. However, we illustrate the use and utility of our methods using the average results, within which there are several points of interest.

The most obvious point is that using the proxy object to mediate data transfer between client and server significantly impacts performance, reducing it to approximate 10 percent of that for the simple client-server pattern. Some impact is certainly expected due to the use of three concurrent client-server pairs, and reduction to 30 percent of the single pair performance would be plausible. Clearly, using a proxy object has a significant additional impact on performance. While not particularly surprising, this result emphasizes the importance of application scenario based testing. This pattern was, for example, discussed in a popular magazine [22] and is used by one of our colleagues as the basis for a WWW meta-search engine. Clearly, any developer contemplating such an architecture would be grateful to know the likely impact before implementing the software.

The second point of interest is that both TAO and OmniORB enjoy a significant performance increase in moving from Linux to Solaris, while the TAO performance for the client-server pattern was relatively constant between the two systems. A third significant observation is that the magnitude of the performance increase for OmniORB in moving from Linux to Solaris is much greater, increasing three to five fold in most cases. Finally, the difference between TAO and CORBAplus performance under Solaris is greater under this pattern than under the client-server pattern.

These observations support our assertion that application performance scenarios, performance patterns, should be part of any comprehensive benchmark. The comparative performance between different ORBs on the same operating system and between the same ORB on different operating systems changed significantly with the change in pattern. This also supports our idea that developers using performance results to select an ORB and operating system as an implementation platform should use test results for object architectures, performance patterns, which faithfully represent their proposed application.

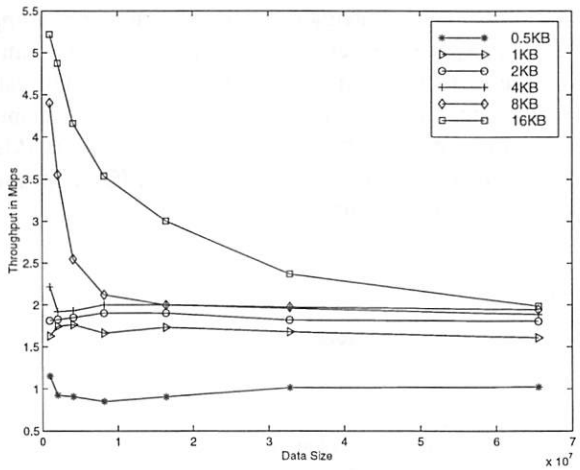


Figure 12. Proxy Throughput for TAO on Solaris

We also changed the test type, as we did for the client-server pattern, to test the *throughput* performance among the object pairs. Figure 11 shows results for the OmniORB under Solaris while Figure 12 presents the throughput results for TAO. Both tests show throughput is reduced from five to ten fold. The TAO results still show an orderly increase in throughput with buffer size, although this converges to a level of 2 Mb/s for all but the smallest buffer size. OmniORB performance, in contrast, does not vary in nearly as orderly a manner with buffer or data set size, and does not converge to similar throughput for most buffer sizes. The performance using 8 KB, 4 KB, and 2 KB buffers is particularly interesting. As with the client-server pattern, the 4 KB buffer size provides the best performance, but 8 KB buffers do substantially better for small data set than large. This could easily be due to system level buffering effects.

Determining why the throughput varies in these ways with data and buffer size will require gathering information from the operating system layer to see if the networking protocols play a role, and gathering information from the ORB layer to see if there is an influence at that level. Section 4.3 illustrates how we might use the DSKI to gather protocol layer information, but discusses a simpler example.

4.3 Using the DSKI

This section briefly illustrates the use of the DSKI to gather performance information from the operating system during a test using the simple client-server pattern

discussed in Section 4.1. As discussed in Section 3 the DSKI creates a stream of time stamped records for each occurrence of a predefined event in the operating system kernel. The set of event records produced can be post-processed to calculate the time spent in providing different types of system services. In this case, the time spent in various portions of the TCP/IP stack can be calculated because we have defined a set of events capable of tracing the progress of packets through the TCP/IP stack. Figure 13 presents the NetSpec PMO script implementing the experiment.

```
cluster {
  corba testbed2 {
    NameOfORB  = omniORB2;
    TypeOftest = rrstring;
    TestParams = (
      numsamples = 1, minsize    = 1,
      maxsize    = 2, multiples  = 2,
      predelay   = 3, postdelay  = 3,
      duration   = 30 );
    protocol   = iiop;
    objname    = omni-receiver;
    role       = server;
    relations  = client(omni-sender);
    criteria   = throughput;
    qos        = normal;
    own        = testbed2(interface = eth,
                          port = 41777);
  }

  corba testbed1 {
    NameOfORB  = omniORB2;
    TypeOftest = rrstring;
    TestParams = (
      numsamples = 1, minsize    = 1,
      maxsize    = 2, multiples  = 2,
      predelay   = 3, postdelay  = 3,
      duration   = 30 );
    protocol   = iiop;
    objname    = omni-sender;
    role       = client;
    relations  = server(omni-receiver);
    criteria   = throughput;
    qos        = normal;
    own        = testbed1 (interface = eth);
  }

  dstream testbed1 {
    type = active (numevents = 100, port=40778,
                                    duration=30);
    ds_tcpip = all;
  }

  dstream testbed2 {
    type = active (numevents = 100, port=40778,
                                    duration=30);
    ds_tcpip_read = all;
  }
}
```

Figure 13. PMO NetSpec Script using DSKI

Figure 14 presents the packet flow in and out of the Sender and Receiver hosts, as well as the performance figures from the kernel obtained using the DSKI. The socket, TCP, IP, and Ethernet layers are numbered 1 through 4 on the sending host, and 5 through 8 from the bottom up on the receiver host. The path of a packet sent

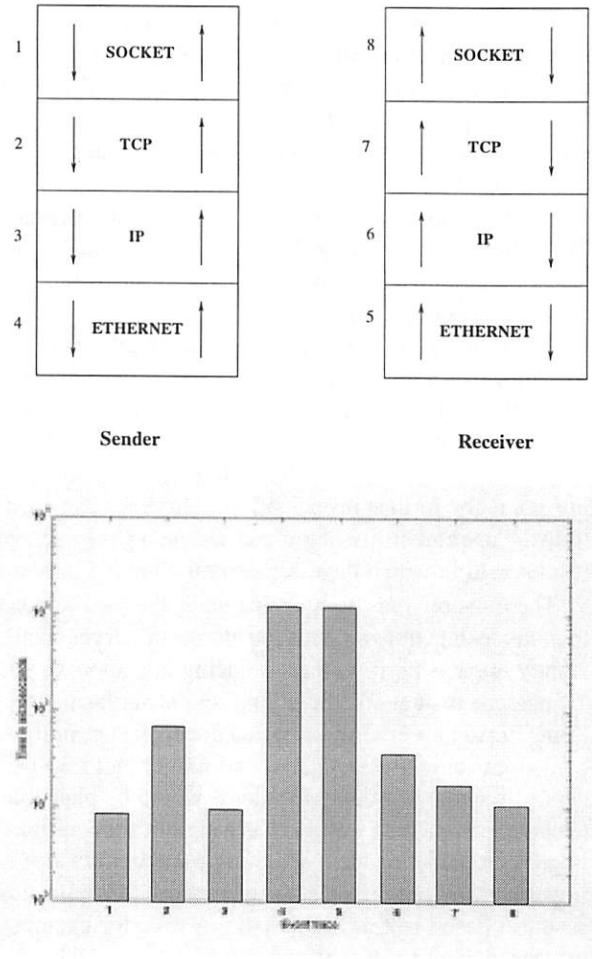


Figure 14. Time trace in the Operating System Layers Obtained Using DSKI

from the sender to the receiver in the diagram would thus flow through layers in numerical order. The bar graph in Figure 14 shows the time in microseconds spent by a packet in each layer, by number.

This experiment demonstrates that the DSKI provides the ability to gather fine grain operating system level time stamped events. Note that the Y-axis in Figure 14 is a logarithmic scale, showing that we were able to monitor time intervals ranging from 10 microseconds to 10 milliseconds. It also illustrates an important feature of the DSKI which we call *tagging*. Each event in the operating system can include a context dependent *tag* when it logs the event. In the case of tracing TCP/IP performance we used the port and sequence numbers to uniquely identify each packet as it moves among protocol layers on each machine.

A similar approach would be used to investigate how buffer size affects throughput under the proxy pattern. DSKI events could be used to monitor when and how

packets are transmitted, combined or broken into fragments, and what size system buffers are used to hold them. Tagging would be used to determine the progress of each packet, and would tell us if the buffer size influenced protocol behavior. Monitoring at the system level might also tell us something about the middleware, even without explicit instrumentation. If, for example, buffers of one size were given to the CORBA layer, but buffers of a different size were passed onto the system layer we would know that the middleware was manipulating the data buffers.

The DSKI can thus be used to investigate a wide range of interactions between the operating system and software layers using its services. This is by no means a simple endeavor, since it requires access to the system source code and sufficient knowledge of the system to enable the investigator to define a reasonable set of events, and then to interpret the results. However, for the investigator willing to learn how to use it well, it can provide an important new source of detailed information from the operating system layer which plays an important role in determining what aspects of endsystem architecture limit application performance under various sets of execution conditions.

5 Conclusions and Future Work

The performance of CORBA based applications implemented as sets of objects is greatly influenced by by the *application context* and by the performance of the ORB *endsystem*. Application developers need to evaluate how candidate application object architectures will perform within heterogenous computing environments, but a lack of standard and user extendable performance benchmark suites exercising all aspects of the ORB endsystem under realistic application scenarios makes this difficult. This paper introduced the *Performance Pattern Language* and the *Performance Measurement Object* which address these problems by providing, under NetSpec control, an automated script based framework within which extensive ORB endsystem performance benchmarks may be efficiently described and automatically executed.

The tools described are implemented, and the viability of the framework they provide has been demonstrated by implementation of small but non-trivial sets of performance evaluation scripts. The examples presented show that the full range of evaluation information can be gathered and a rich set of performance scenarios examined. The automated nature of the script driven framework is also important because it makes it possible to describe and conduct a large set of evaluation experiments covering a adequately diverse and detailed set of scenarios and performance metrics.

Performance evaluation of CORBA based distributed applications, and of candidate object architectures, is an extremely important and difficult problem. Current benchmarking and testing methods are not as comprehensive as they might be because the scale and complexity required is daunting. The tools described here make a significant increase in the scale, complexity, and level of detail of performance evaluation studies possible, thus significantly advancing the state of the art.

Our future work will include creation of new test types and performance patterns. We are particularly interested in extending this approach to testing to include execution of applications under real-time constraints. We will use this set of tests to drive an investigation of what kinds of system support can be used to improve real-time performance of ORB based applications. We will concentrate on a time constrained event service and integration of operating system scheduling, I/O, and ORB level operations to improve time constrained communication among objects.

Availability

NetSpec and many daemons developed for various types of performance evaluation are publicly available. The PMO and the PPL have been developed with support from Sprint, and are not yet publicly available, but should be soon. The work described here is intended as a contribution to the CORBA community and are intended for full availability on the WWW. For further details check:

`www.ittc.ukans.edu/~niehaus/research.html`

References

[1] Buchanan, B., Niehaus, D., Sheth, S., and Wijata, Y. The Data Stream Kernel Interface. Technical Report ITTC-FY98-TR-11510-04, University Of Kansas, December 1997.

[2] CORBA Comparison Project. http://nenya.ms.mff.cuni.cz/thegroup/COMP/index.html.

[3] DeWitt, D. J. The Wisconsin Benchmark: Past, present, and Future. In *The Benchmark Handbook for Database and Transaction Processing Systems*, Ed. by Jim Gray, Morgan Kaufmann, Inc., 1993, pp. 269-316.

[4] Eide, E., Frei, K., Ford, B., Lepreau J. and Lindstorm, G. Flick: A Flexible, Optimizing IDL Compiler. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, June 1997.

[5] Expersoft Corporation Corbaplus. http://www.expersoft.com/.

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison-Wesley, 1994.

[7] Gokhale, A., and Schmidt, D. C. Measuring the Performance of Communication Middleware on High-Speed Networks. In *SIGCOMM*, August 1996.

[8] Gokhale, A. and Schmidt, D. C. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *ICDCS*, 1997.

[9] Gopinath, A. Performance Measurement and Analysis of Real-time CORBA Endsystems. Master's thesis, University Of Kansas, June 1998.

[10] Gopinath, A., Nimmagadda, S., Liyanaarachchi, C. and Niehaus, D. Performance Measurement Of CORBA Endsystems. Technical Report ITTC-FY99-TR-14120-01, University Of Kansas, june 1998.

[11] Inter Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[12] Lee, B. O., Frost, V. S., and Jonkman, R. Netspec 3.0 Source Models for telnet, ftp, voice, video and www Traffic, January 1997.

[13] MCITT. http://www.mel.nist.gov/msidstaff/flater/mcitt.

[14] Mowbray, T. J. and Malveau, R. C. *CORBA Design Patterns*. John Wiley & Sons Inc., 1997.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification v 2.2, February 1998.

[16] Roel Jonkman . Netspec: Philosophy, Design and Implementation. Master's thesis, University of Kansas, Lawrence, Kansas, February 1998.

[17] Schmidt, D. C., Bector, R., Levine, D. L., Mungee, S. and Parulkar, G. TAO: A Middleware Framework for Real-Time ORB Endsystems. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.

[18] Schmidt, D. C., Mungeee, S., Flores-Gaitan, S., and Gokhale, A. Alleviating priority inversion and non-determinism in real-time corba orb core architectures. In *Real Time Applications Symposium*, June 1998.

[19] Schmidt, D. C. Evaluating Architectures for Multi-threaded Object Request Brokers. *Communications of the ACM*, 41(10):54–60, 1998.

[20] Schmidt, D. C., Levine, D. L., and Mungee, S. The Design and Perfromance of Real-Time Object Request Brokers. In *Computer Communications*, volume 21, pages 294–324, April 1998.

[21] Schmidt, D. C., Gokhale, A., Harrison, T. H. and Parulkar, G. A High Performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[22] Smith, A. Distributed obsever chains. *Java Report*, 3(9):49–58, 1998.

[23] The Olivetti & Oracle Research Laboratory. Omniorb2, http://www.orl.co.uk/omniorb/.

[24] The Open Group. http://tetworks.opengroup.org/datasheet.html.

[25] Turbyfill, C., Orji, C. and Bitton, D. AS3AP - A Comparative Relational Database Benchmark, In *Proceedings of the IEEE COMPCON*, 1989, pp. 560-564.

[26] Turbyfill, C., Orji, C. and Bitton, D. AS3AP: An ANSI SQL Standard Scaleable and Portable Benchmark for Relational Database Systems, In *The Benchmark Handbook for Database and Transaction Processing Systems*, Ed. by Jim Gray, MorgaKaufmann, Inc., 1993, pp. 317-358.

[27] Vinoski, S. CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[28] X/Open Company Ltd. VSORB Test Suite release 1.0.0, April 1997.

Object–Oriented Pattern–Based Parallel Programming with Automatically Generated Frameworks

Steve MacDonald, Duane Szafron, and Jonathan Schaeffer
Department of Computing Science, University of Alberta
{stevem,duane,jonathan}@cs.ualberta.ca, http://www.cs.ualberta.ca/~{stevem,duane,jonathan}

Abstract

The CO_2P_3S parallel programming system uses design patterns and object–oriented programming to reduce the complexities of parallel programming. The system generates correct frameworks from pattern template specifications and provides a layered programming model to address both the problems of correctness and openness. This paper describes the highest level of abstraction in CO_2P_3S, using two example programs to demonstrate the programming model and the supported patterns. Further, we introduce *phased parallel design patterns*, a new class of patterns that allow temporal phase relationships in a parallel program to be specified, and provide two patterns in this class. Our results show that the frameworks can be used to quickly implement parallel programs, reusing sequential code where possible. The resulting parallel programs provide substantial performance gains over their sequential counterparts.

1 Introduction

Parallel programming offers potentially substantial performance benefits to computationally intensive applications. Using additional processors can increase the computational power that can be applied to large problems. Unfortunately, it is difficult to use this increased computational power effectively as parallel programming introduces new complexities to normal sequential programming. The programmer must create and coordinate concurrency. Synchronization may be necessary to ensure data is used consistently and to produce correct results. Programs may be nondeterministic, hampering debugging by making it difficult to reproduce error conditions.

We need development tools and programming techniques to reduce this added programming complexity. One such tool is a parallel programming system (PPS). A PPS can deal with complexity in several ways. It can provide a programming model that removes some of the added complexity. It can also provide a complete tool set for developing, debugging, and tuning programs. However, it is likely that there will still be some additional complexity the user must contend with when writing parallel programs, even with a PPS. We can ease this complexity by using new programming techniques, such as design patterns and object–oriented programming. Design patterns can help by documenting working design solutions that can be applied in a variety of contexts. Object–oriented programming has proven successful at reducing the software effort in sequential programming through the use of techniques such as encapsulation and code reuse. We want to apply these benefits to the more complex domain of parallel programming.

The CO_2P_3S[1] parallel programming system supports pattern–based parallel program development through framework generation and multiple layers of abstraction [8]. The system can be used to parallelize existing sequential code or write new, explicitly parallel programs. This system automates the use of a selected set of patterns through *pattern templates*, an intermediary form between a pattern and a framework. These pattern templates represent a pattern where some of the design alternatives are fixed and others are left as user–specified parameters. Once the template has been fully specified, CO_2P_3S generates a framework that implements the pattern in the context of the fixed and user–supplied design parameters. Within this framework, we introduce sequential *hook methods* that the user can implement to insert application–specific functionality. In the CO_2P_3S programming model, the higher levels of abstraction emphasize correctness and reduce the probability of programmer errors by providing the parallel structure and synchronization in the framework such that they cannot be modified by the user. The lower layers emphasize *openness* [14], gradually exposing low–level implementation details and introducing more opportunities for per-

[1] Correct Object–Oriented Pattern–based Parallel Programming System, pronounced "cops".

formance tuning. The user can work at an appropriate level of abstraction based on what is being tuned.

The key to reducing programmer errors is the decomposition of the generated framework into parallel and sequential portions. The framework implements both the parallel structure of the program and the synchronization for the execution of the hook methods. Neither of these attributes of the program can be modified at the highest level of abstraction. This decomposition allows the hook methods to be implemented as sequential methods, so users can concentrate on implementing applications rather than worrying about the details of the parallelism.

We introduce *phased parallel design patterns* in this paper. Phased patterns are unique in that they express a temporal relationship between different phases of a parallel program. Although all patterns have temporal aspects (such as specific sequences of method invocations), the intent of phased patterns is to deal with changing concurrency requirements for different phases of a parallel program. We introduce two such patterns, the Method Sequence and the Distributor, both of which are new. The Method Sequence can be used to implement phased algorithms, explicitly differentiating between the different phases of a parallel algorithm. This pattern recognizes that efficiently parallelizing a large program will likely require the application of several parallel design patterns. The Distributor pattern allows the user to selectively parallelize a subset of methods on an object, acknowledging that not all operations may have sufficient granularity for parallel execution.

We also introduce a structural pattern, the Two–Dimensional Mesh. The Mesh was written in an object–oriented fashion to fit within the CO_2P_3S system.

To demonstrate the use of the CO_2P_3S system, we present the development of two example programs. Our first example, reaction–diffusion texture generation [17], uses the Mesh pattern to simulate the reaction and diffusion of two chemicals over a two–dimensional surface. The second program implements the parallel sorting by regular sampling algorithm [13] using the Method Sequence and Distributor patterns. Both programs are implemented using the facilities provided at the highest level of abstraction in CO_2P_3S to demonstrate the utility of our patterns and the utility of the frameworks we generate to support our pattern templates.

The research contributions of this paper are:

- A layered parallel programming model that presents several different programming abstractions to the user. Each layer emphasizes different concerns, starting with program correctness at the highest level of abstraction, and providing openness and performance tuning at lower levels.
- The generation of a correct parallel framework from a pattern template specification, coupled with correctness guarantees by preventing the user from modifying the structure of the framework at the highest level of abstraction.
- An easy–to–use programming model that allows users to implement a parallel program by writing a small amount of sequential code to reuse their existing application code.
- An object–oriented pattern–based tool for parallel programming. We also introduce a new type of parallel design pattern, called phased patterns, to express time–related aspects of a parallel program.
- A demonstration of the benefits of the CO_2P_3S system using two example programs. These examples illustrate the use of the highest level of abstraction in the CO_2P_3S model and demonstrate the benefits of using a high–level tool for parallel programming.

2 Overview of the CO_2P_3S System

This section presents a brief overview of the CO_2P_3S system. A more detailed description can be found in [8].

The CO_2P_3S parallel programming system provides three levels of abstraction that can be used in the development of parallel programs. These abstractions provide a programming model that allows the programs to be tuned incrementally, allowing the user to develop parallel programs where performance is more directly commensurate with effort. These abstractions, from highest to lowest, are:

Patterns Layer The user selects a parallel design pattern template from a palette of supported templates. These templates represent a partially specified design pattern, where some of the design tradeoffs have been fixed. Each pattern template has several parameters that must be supplied before the template can be instantiated, allowing the user to specialize the framework implementing the pattern for its intended use. Instantiating the pattern template generates code that forms a framework for the template. The code consists of one or more

abstract classes that implement the parallel structure of the pattern template together with concrete subclasses, as well as any required collaborator classes. The framework code is customized in two ways: application–specific pattern template parameters and user–supplied implementations of specific sequential hook methods in the concrete subclasses (using the Template Method design pattern [4]). Since the user cannot modify the parallel structure at this layer, parallel correctness is ensured. A complete program consists of either a single framework or several frameworks composed together.

Intermediate Code Layer This layer provides a high–level, object–oriented, explicitly–parallel programming language, a superset of an existing OO language. The user manipulates both parallel structural code and application–specific code using this intermediate language.

Native Code layer The intermediate language is transformed into code for a native object–oriented language (such as Java or C++). This code provides all libraries used to implement the intermediate code from the previous layer. The user is free to use the provided libraries and language facilities to modify the program in any way.

Users can move down through the different abstractions, selecting a suitable layer based on the desired performance of their applications or on how comfortable they are with a given abstraction.

Several critical aspects of the framework are demonstrated by the use of hook methods for introducing application–specific functionality. First, the parallel structural code cannot be modified in the Patterns Layer, which allows us to make correctness guarantees about the parallel structure of the program. Second, it allows users to concentrate on implementing their applications without worrying about the structure of the parallelism. Also, by ensuring that the structural code provides proper synchronization around hook method invocations, the user can write sequential code without having to take into account the parallelism provided by the framework. Lastly, we provide suitable default implementations of the hook methods in the abstract classes of the framework. These default methods permit the framework to be compiled and executed immediately after it is generated, without implementing any of the hook methods. The program will execute with a simple default behaviour. This provides users with a correct implementation of the structure of the pattern before they begin adding the hook methods and tuning the program.

The CO_2P_3S system currently supports several parallel design patterns through its templates, which also use a group of sequential design patterns. These patterns are written specifically for solving design problems in the parallel programming domain. The patterns used in this paper are:

Method Sequence This new pattern supports the creation of phased algorithms by invoking an ordered sequence of methods on a Facade [4] object.

Distributor This new pattern supports data–parallel style computations by forwarding a method from a parent object to a fixed number of child objects, all executing in parallel.

Two–Dimensional Mesh This pattern supports iterative computations for a rectangular two–dimensional mesh, where a surface is decomposed into a set of regular, rectangular partitions.

A more detailed description of these patterns, along with the other patterns supported by CO_2P_3S, can be found in our pattern catalogue [7].

The context of our patterns is the architecture and programming model of CO_2P_3S. Therefore, we have made several changes to the structure of the basic design pattern documentation [4]. Since our patterns are for the parallel programming domain, the pattern description includes concurrency–related specifications, such as synchronization and the creation of concurrent activity. From the pattern, we produce a CO_2P_3S pattern template and an example framework, which we also describe in the pattern document. While these two sections are not strictly pattern specifications, they illustrate the use of the pattern while documenting the CO_2P_3S templates and frameworks. So while we have added CO_2P_3S–specific sections to our pattern descriptions, we have preserved the instructional nature of design patterns.

We note that our patterns still represent abstract design solutions. From the abstract pattern, we fix some of the design tradeoffs and allow the user to specify the remaining tradeoffs using parameters. This intermediate form, which is less abstract than a pattern but less concrete than a framework, is called a pattern template. We use a fully specified pattern template to generate parametrically related families of frameworks, with each framework in the family implementing the same basic pattern structure but specialized with the user–supplied parameters. However, the pattern templates are a design construct used to specify a framework; the templates do not contain nor provide code themselves. The generated

frameworks provide reusable and extendible code implementing a specific version of the pattern. These frameworks take into account our decomposition of a program into its sequential and parallel aspects. The ability of the user to modify a framework is dictated by the level of abstraction that the user is using.

In creating the pattern templates, we try to fix as few of the design tradeoffs as possible. However, in fixing any part of the design, we will create situations in which it will be necessary to modify certain elements of a program to fit within the limitations we impose. For the example programs that follow, we discuss the design in terms of the patterns, and discuss the implementation in terms of the pattern templates and the frameworks. The implementation of the programs may need to be modified from the initial design to account for the fixed design tradeoffs in the pattern templates and frameworks.

3 Example Applications

In this section, we detail the design and implementation of two example programs. These examples demonstrate the applicability of our patterns to application design, and show how the pattern templates and generated frameworks can be used to implement the programs. The first example uses a reaction–diffusion texture generation program to show how we have reworked the Mesh pattern in an object–oriented fashion. The second example, parallel sorting by regular sampling, uses the Method Sequence and Distributor patterns. We also use this example to highlight the temporal nature of these two patterns and to show the composition of CO_2P_3S frameworks. The implementation of these two examples demonstrates that the Patterns Layer of CO_2P_3S can be used to write parallel programs with good performance. It also shows the benefits of a high–level object–oriented tool for parallel programming.

3.1 Reaction–Diffusion Texture Generation

Reaction–diffusion texture generation [17] can be described as two interacting LaPlace equations that simulate the reaction and diffusion of two chemicals (called *morphogens*) over a two–dimensional surface. This simulation, started with random concentrations of each morphogen on the surface, can produce texture maps that approximate zebra stripes. The problem uses Gauss–Jacobian iterations (each iteration uses the results from the previous iteration to compute the new values, without any successive overrelaxation) and is solved using straightforward convolution. The simulation typically executes until the total change in morphogen concentration over the surface falls below some threshold. We use a fully toroidal mesh, allowing the morphogens to wrap around the edges of the surface. The toroidal boundary conditions ensure that the resulting texture can be tiled on a display without any noticeable edges between tiles.

3.1.1 Parallel Implementation

An obvious approach to this problem is to decompose the two–dimensional surface into regular rectangular regions and to work on these regions in parallel. Our solution must be more complex because we cannot evaluate each region in isolation. Each point on the surface needs the concentration values from its neighbours to calculate its new value, so points on the edge of a region need data from adjoining regions. Thus, we require an exchange of region boundaries between iterations. Further, Gauss–Jacobian iterations introduces dependencies between iterations that need to be addressed by our parallel implementation of this problem.

3.1.2 Implementation in CO_2P_3S

Design and Pattern Specification The first step in implementing a CO_2P_3S program is to analyze the problem and select the appropriate design pattern. This process still represents the bottleneck in the design of any program. Given the requirements of our problem, the Mesh pattern is a good choice for several reasons. The problem is an iterative algorithm executed over a two–dimensional surface. Further, our approach is to decompose the surface into regular rectangular regions, a decomposition that is automatically handled by the framework for the Mesh pattern template.

The Mesh pattern consists of two types of objects, a collector object and a group of mesh objects. The structure of the Mesh is given in Figure 1. The collector object is responsible for creating the mesh objects, distributing the input state over the mesh objects, controlling the execution of the mesh objects, and collecting the results of the computation. The mesh objects implement an iterative mesh computation, a loop that exchanges boundaries with neighbouring mesh elements and computes the new values for its local mesh elements. The iterations continue until all the mesh objects have finished

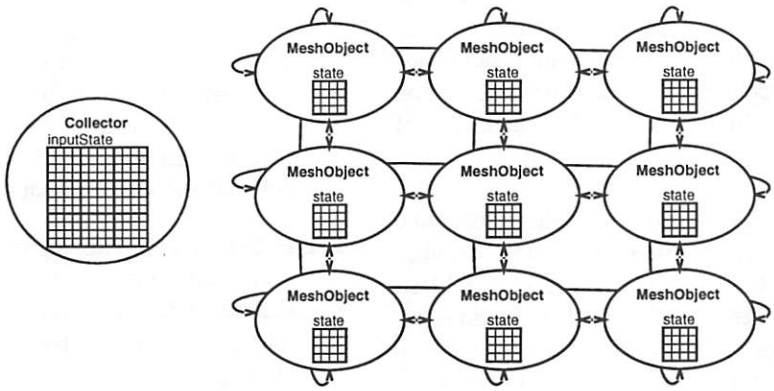


Figure 1: An object diagram for the structure of the Mesh pattern. The arcs indicate object references. The `Collector` object has references to all of the instances of `MeshObject`, but the arcs are omitted for clarity.

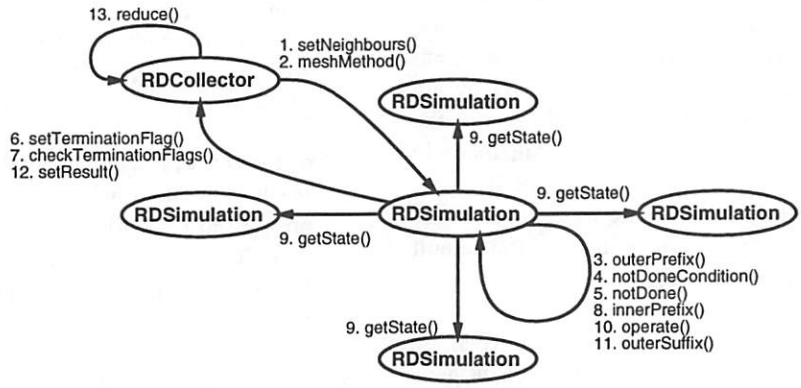


Figure 2: The method invocations in the Mesh implementation of the reaction–diffusion problem. The methods are executed in the order in which they are numbered. Calls 4 through 10 are the main loop of the mesh computation and are repeated until the computation has completed.

computing, when all of the morphogen concentrations in all of the mesh objects have stabilized in our example.

This pattern requires synchronization because of dependencies between iterations. Specifically, we need to ensure that the boundary elements for each mesh element have all been computed before exchanging them, to prevent a neighbour from using incorrect data in its next iteration. We also require synchronization for determining the termination condition, since all of the mesh objects must cooperate to decide if they have finished. This synchronization is necessarily pessimistic to handle the general case. Individual programs using this pattern may remove any unnecessary synchronization.

The Mesh pattern is not specific to the reaction–diffusion example. It can be used to implement other finite element calculations and image processing applications.

Once the pattern has been selected, the user selects the corresponding pattern template and fills in its parameters. For all pattern templates, the names of both the abstract and concrete classes for each class in the resulting framework are required. In CO_2P_3S, the user only specifies the concrete class names; the corresponding abstract class names are prepended with "Abstract". For our Mesh example, we specify `RDCollector` for the collector class and `RDSimulation` for the mesh class, which also creates the abstract classes `AbstractRDCollector` and `AbstractRDSimulation`. We further specify the type of the two–dimensional array that will be distributed over the mesh objects, which is `MorphogenPair` for the reaction–diffusion example. Finally, we specify the boundary conditions of the mesh, which is set to fully toroidal (where each mesh object has all four neighbours by wrapping around the edges of the mesh, as shown in Figure 1). We can select other boundary conditions (horizontal–toroidal, vertical–toroidal, and non–toroidal); we will see the ef-

fects of different conditions later in this section. The dimensions of the mesh are specified via constructor arguments to the framework. The input data is automatically block–distributed over the mesh objects, based on the dimensions of the input data and the dimensions of the mesh.

Using the Framework From the pattern template specification, the CO_2P_3S system generates a framework implementing the specific instance of the Mesh pattern given the pattern template and its parameters. The framework consists of the four classes given above with some additional collaborator classes. The sequence of method calls for the framework is given in Figure 2.

Once the framework is generated, the user can implement hook methods to insert application–specific functionality at key points in the structure of the framework. The selection of hook methods is important since we enforce program correctness at the Patterns Layer by not allowing the user to modify the structural code of the framework. The user implements the hook methods in the concrete class by overriding the default method provided in the abstract superclass in the framework. If the hook method is not needed in the application, the default implementation can be inherited.

To demonstrate the use of the hook methods, we show the main execution loop of the Mesh framework, as generated by CO_2P_3S, in Figure 3. The hook methods are indicated in bold italics. There are hook methods for both the collector and mesh objects in the Mesh framework. For the collector, the only hook method is:

`reduce()` This method, invoked after the mesh computation has finished, allows the user to perform a reduction on the results. By default, this method returns the input array updated with the results of the mesh computation.

The hook methods for the mesh objects are:

`outerPrefix()` This method is invoked before the mesh computation is started. It can be used for initializing the mesh object. By default, this method does nothing.

`notDone()` This method is used to determine if the mesh object has finished its local computation. Note that the computation finishes only when all mesh objects have finished. By default, this method returns `false`, indicating that the mesh object has finished its computation. This method is not directly invoked from the `meshMethod()` method. It is invoked indirectly from the `notDoneCondition()` method, which uses the result of this hook method in the `setTerminationFlag()` method to set the termination status of the current mesh object and then uses `checkTerminationFlags()` to calculate the global termination condition.

`innerPrefix()` This method is invoked first in the mesh computation loop, before the boundary exchange. It can be used for any precomputations required in the loop. By default, this method does nothing.

The operation methods These methods implement the mesh computation. There are nine methods that may be used depending on the boundary conditions. These are described below. By default, these methods do nothing. They are invoked indirectly from the `operate()` method. These methods replace the `innerSuffix()` method.

`outerSuffix()` This method is invoked after the mesh computation has finished but before results are passed back to the collector object. It can be used for any post–processing or cleanup required by the mesh object. By default, this method does nothing.

The implementation of the `operate()` method called in the code from Figure 3 invokes a subset of the nine operation methods given in Figure 4. The boundary conditions and the position of the mesh object determine which of the operation methods are used. For instance, consider the two meshes in Figure 5. For the fully toroidal mesh in Figure 5(a), there are no boundary conditions. Thus, only the `interiorNode()` hook method is invoked. For the horizontal–toroidal mesh in Figure 5(b), there are three different cases, one for each row. The mesh objects in the different rows, from top to bottom, invoke the `topEdge()`, `interiorNode()`, and `bottomEdge()` hook methods for the mesh operation. The implementation of the `operate()` method uses a Strategy pattern [4], where the strategy corresponds to the selected boundary conditions. This strategy is a collaborator class generated with the rest of the framework. It is also responsible for setting the neighbours of the mesh elements after they are created, using the `setNeighbours()` method (from Figure 2). At the Patterns Layer, the user does not modify this class. Each mesh object automatically executes the correct methods, depending on its location in the mesh.

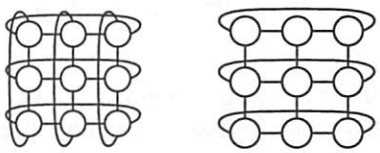
Now we implement the reaction–diffusion texture generation program using the generated Mesh framework.

```
public void meshMethod()
{
    this.outerPrefix() ;
    while(this.notDoneCondition()) {
        this.innerPrefix() ;
        MorphogenPair[][] leftState = left.getState() ;
        MorphogenPair[][] rightState = right.getState() ;
        MorphogenPair[][] upState = up.getState() ;
        MorphogenPair[][] downState = down.getState() ;
        this.operate(leftState, rightState, upState, downState) ;
    } /* while */
    this.outerSuffix() ;
    this.getCollector().setResult(this.getState()) ;
} /* meshMethod */
```

Figure 3: The main execution loop of a mesh. Hook methods are shown in bold italics.

```
void topLeftCorner(MorphogenPair[][] right, MorphogenPair[][] down) ;
void topEdge(MorphogenPair[][] right, MorphogenPair[][] left,
             MorphogenPair[][] down) ;
void topRightCorner(MorphogenPair[][] left, MorphogenPair[][] down) ;
void leftEdge(MorphogenPair[][] right, MorphogenPair[][] up,
             MorphogenPair[][] down) ;
void interiorNode(MorphogenPair[][] left, MorphogenPair[][] right,
             MorphogenPair[][] up, MorphogenPair[][] down) ;
void rightEdge(MorphogenPair[][] left, MorphogenPair[][] up,
             MorphogenPair[][] down) ;
void bottomLeftCorner(MorphogenPair[][] right, MorphogenPair[][] up) ;
void bottomEdge(MorphogenPair[][] left, MorphogenPair[][] right,
             MorphogenPair[][] up) ;
void bottomRightCorner(MorphogenPair[][] left, MorphogenPair[][] up) ;
```

Figure 4: The hook methods for the mesh operations.



(a) A fully toroidal mesh.

(b) A horizontal–toroidal mesh.

Figure 5: Example mesh structures.

First, we note that we do not need a reduction method, as the result of the computation is the surface computed by each region. Also, we do not require the `outerPrefix()` or the `outerSuffix()` methods. The `innerPrefix()` method is required because we have chosen to keep two copies of each morphogen array, a read copy for getting previous data and a write copy for update during an iteration. This scheme uses additional memory, but obviates the need to copy the array in each iteration. Each iteration must alternate between using the read and write copies, which is accomplished by reversing the references to the arrays in the `innerPrefix()` method. Given our fully toroidal mesh, we only need to implement the mesh operation in the `interiorNode()` hook method.

The `notDone()` method checks the local mesh state for convergence. Each mesh object returns a Boolean flag indicating if it has finished its local computation, and these flags are used to determine if all of the mesh objects have finished. The pattern template fixes the flags as Booleans, which does not allow the global termination conditions given in Section 3.1 to be implemented. Instead, our simulation ends when the change in morphogen concentration in each cell falls below a threshold. Although this restriction forced us to modify this program, it simplifies the pattern template specification and reduces the number of hook methods. This termination behaviour can be modified at the Intermediate Code Layer if a global condition must be implemented.

After completing the specification and implementation of the Mesh framework, the user must implement the code to instantiate the objects and use the framework. The Java code is given in Figure 6, where we use constants for the width and height of the data and the mesh, but these values could be obtained dynamically at run–time from a file or from the user.

3.1.3 Evaluation

In this section, we evaluate the Patterns Layer of CO_2P_3S using the reaction–diffusion texture generator. Our basis for evaluation is the amount of code written by the user to implement the parallel program and the run–time performance. These results are based on a Java implementation of the problem.

In the following discussion, we do not include any comments in counts of lines of code. All method invocations and assignments are considered one line, and we count all method signatures (although the method signatures for the hook methods are generated for the user).

The sequential version of the reaction–diffusion program was 568 lines of Java code. The complete parallel version, including the generated framework and collaborating classes, came to 1143 lines. Of that 1143 lines, the user wrote (coincidentally) 568 lines, just under half. However, 516 lines of this code was taken directly from the sequential version. This reused code consisted of the classes implementing the morphogens. This morphogen code had to be modified to use data obtained from the boundary exchange, whereas the sequential version only used local data. This modification required one method to be removed from the sequential version and several methods added, adding a total of 52 lines of code to the application. The only code that could not be reused from the sequential version was the mainline program. In addition, the user was required to implement the hook methods in the Mesh framework. These methods were delegated to the morphogen classes and required only a single line of code each.

We note that this case is almost optimal; the structure of the simulation was parallelized without modifying the computation. Also, the structure of the parallel program is close to the sequential algorithm, which is not always the case. These characteristics allowed almost all of the sequential code to be reused in the parallel version.

This program was executed using a native–threaded Java implementation from SGI (Java Development Environment 3.1.1, using Java 1.1.6). The programs were compiled with optimizations on and executed on an SGI Origin 2000 with 42 195MHz R10000 processors and 10GB of RAM. The Java virtual machine was started with 512MB of heap space. Results were collected for programs executed with the just–in–time (JIT) compiler enabled and then again with the JIT compiler disabled. Disabling the JIT compiler effectively allows us to observe how the frameworks behave on problems with increased granularity. Speedups are based on wall–clock time and are compared against a sequential implementation of the same problem executed using a green threads virtual machine. Note that the timings are only for the mesh computation; neither initialization of the surface nor output is included. The results are given in Table 1.

With the JIT enabled, the speedups for the program tail off quickly. As we add more processors, the granularity of the mesh computation loop falls and the synchronization hampers performance. The non–JIT version shows good speedups, scaling to 16 processors, showing the effects of increased granularity.

From this example, we can see that our framework promotes the reuse of sequential code; almost all of the morphogen code from the sequential program was reused in the parallel version. This reuse allowed the parallel version of the problem to be implemented with only a few new lines of code (52 lines). The performance of the resulting parallel application is acceptable with the JIT enabled, although the granularity quickly falls.

3.2 Parallel Sorting by Regular Sampling

Parallel sorting by regular sampling (PSRS) is a parallel sorting algorithm that provides a good speedup over a broad range of parallel architectures [13]. This algorithm is explicitly parallel and has no direct sequential counterpart. Its strength lies in its load balancing strategy, which samples the data to generate pivot elements that evenly distribute the data to processors.

The algorithm consists of four phases, illustrated in Figure 7. Each phase must finish before the next phase starts. The phases, executed on p processors, are:

1. In parallel, divide the input array into p contiguous lists and sort each list. Select $p - 1$ evenly spaced sample elements from each sorted list.
2. Select a designated processor to sort the entire set of sample elements. Then, choose $p - 1$ evenly spaced pivot values from the sample set.

```
public static void main(String[] argv)
{
    MorphogenPair[][] data = Main.initializeData(Main.dataWidth,
            Main.dataHeight) ;
    RDCollector collector = new RDCollector(Main.meshWidth,
            Main.meshHeight, data, Main.dataWidth, Main.dataHeight) ;
    /* Start the execution of the simulation.  */
    collector.Execute() ;
    /* Wait for and get the results.  */
    data = (MorphogenPair[][]) collector.getResults() ;
} /* main */
```

Figure 6: The code that starts the reaction–diffusion simulation using the Mesh framework.

Problem Size	JIT disabled				JIT enabled			
	2 proc.	4 proc.	8 proc.	16 proc.	2 proc.	4 proc.	8 proc.	16 proc.
1024×1024	1.96	3.72	6.93	12.39	1.61	2.88	4.49	4.58
	11150 sec	5886 sec	3162 sec	1767 sec	2519 sec	1413 sec	906 sec	887 sec

Table 1: Speedups for the reaction–diffusion example. Wall clock times, rounded to the second, are also provided.

3. In parallel, partition each sorted list into p sublists using the pivot values.

4. In parallel, merge the partitions and store the results back into the array.

3.2.1 Parallel Implementation

The parallelism in this problem is clearly specified in the algorithm from the previous section. We require a series of phases to be executed, where some of those phases use a set of processors computing in parallel. For the parallel phases, a fixed number of processors execute similar code on different portions of the input data.

3.2.2 Phased Parallel Design Patterns

An interesting aspect of the PSRS algorithm is that the parallelism changes in different phases. The first and third phases are similar. The second phase is sequential. Finally, the last phase consists of two subphases (identified below), where each subphase has its own concurrency requirements. We need to ensure that this temporal relationship between the patterns can be expressed. In contrast, other parallel programming systems require the user to build a graph that is the union of all the possible execution paths that are used and leave it to the user to ensure they are used correctly. Alternately, the user must use a pipeline solution, where each pipe stage implements a phase (as in Enterprise [10]). However, the real strength of a pipeline lies in algorithms where multiple requests can be concurrently executing different operations in different pipe stages. Further, a pipeline suggests that a stream of data (or objects in an OO pipeline) is being transformed by a sequence of operations. A phased algorithm may transform its inputs or generate other results, depending on the algorithm.

A further temporal aspect of this algorithm is when to use parallelism. Sometimes we would like to use the same group of objects for both parallel and sequential methods. For instance, some methods may not have enough granularity for parallel execution. Sometimes, as in the second phase of PSRS, we may need to execute a sequential operation on data contained in a parallel framework. This kind of control can be accommodated by adding sequential methods in the generated framework. These methods would use the object structure without the concurrency. In implementing these methods, the user must ensure that they will not interfere with the execution of any concurrent method invocations.

3.2.3 Implementation in CO_2P_3S

Design and Pattern Specification

Method Sequence In the PSRS algorithm, the phases stand out as the first concern. The algorithm suggests four phases. We note, though, that the last phase contains a data dependency. We must ensure that the merging is complete before we can begin writing results back to the original array. Otherwise, the merging phase can

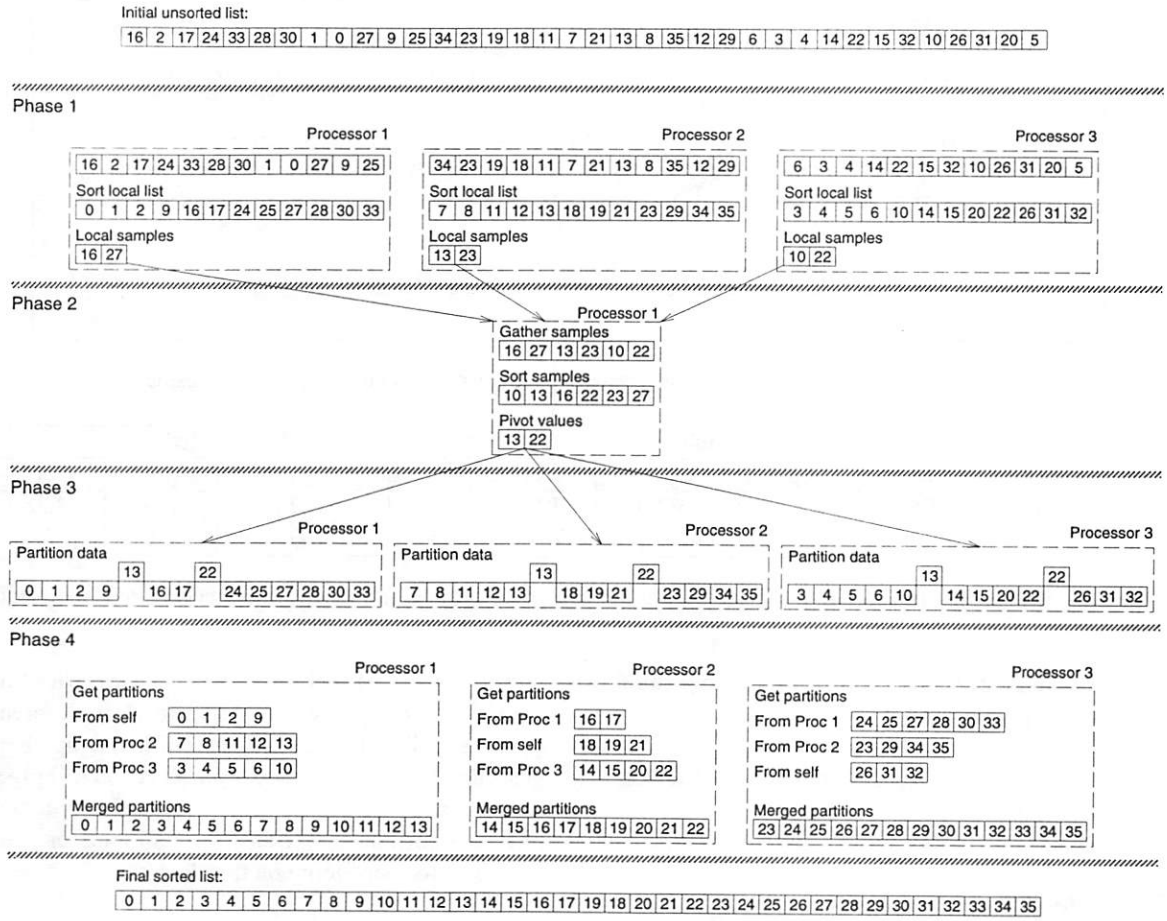


Figure 7: An example of parallel sorting by regular sampling.

read incorrect data. From this observation, we rewrite the fourth phase of PSRS as two subphases:

4.1 Merge the partitions in a temporary buffer.

4.2 Store the results in the original array by copying the temporary buffer.

To implement a series of phases in a program, we can use the Method Sequence pattern. In our example, we have identified two different sets of phases so we apply this pattern twice. The first application of the Method Sequence pattern implements the four phases from the previous section. We apply the pattern again in the implementation of the last phase, to execute the phases identified above. Alternately, we could rewrite the original algorithm as five phases and apply the pattern once. However, our solution is consistent with the original algorithm and also helps us demonstrate the composability of our frameworks in the next subsection.

The Method Sequence pattern is a specialization of the Facade pattern [4] that adds ordering to the methods of the Facade. It consists of two objects, a sequence object and an instance of the Facade pattern [4]. The sequence object holds an ordered list of method names to be invoked on the Facade object. These methods have no arguments or return values. The Facade object supplies a single entry point to the objects that collaborate to execute the different phases. The Facade typically delegates its methods to the correct collaborating object, where these methods implement the different phases for the sequence object. Each phase is executed only after the previous phase has finished. The Facade object is also responsible for keeping any partial results generated by one phase and used in another (such as the pivots generated in the second phase of PSRS and used in the third phase). We include the Facade object for the general case of the Method Sequence pattern, where there may be different collaborating objects implementing different phases of the computation. Without the Facade object, the sequence object would need to manage both the method list and the objects to which the methods are delegated, making the object more complicated.

The Method Sequence pattern has other uses beyond this example. For example, it is applicable to programs that can be written as a series of phases, such as LU factorization (a reduction phase and a back substitution phase).

After designing this part of the program, the user selects the Method Sequence pattern template and fills in the parameters to instantiate the template. For this pattern template, the user specifies the names of the concrete classes for the sequence and Facade classes, and an ordered list of methods to be invoked on the Facade object. Again, the abstract superclasses for both classes have "Abstract" prepended to the concrete class names.

Distributor Now we address the parallelism in the first, third, and last phases. Each of these phases require a fixed amount of concurrency (p processors). If we attempt to vary the number of processors for different phases, we will generate different data distributions that will cause problems for the operations. Further, the processors operate on the same region of data in the first and third phases. If we can distribute the data once to a fixed set of processors, we can avoid redistribution costs and preserve locality. The last phase requires a redistribution of data, but again it must use the same number of processors as used in the previous parallel phases. Similarly, the two subphases for the last phase share a common region, the temporary buffer. It is also necessary for the concurrency to be finished at the end of each phase because of the dependencies between the phases.

Given these requirements, we apply the Distributor pattern. This pattern provides a parent object that internally uses a fixed number of child objects over which data may be distributed. In the PSRS example, the number of children corresponds to the number of processors p. All method invocations are targeted on the parent object, which controls the parallelism. In this pattern, the user specifies a set of methods that can be invoked on all of the children in parallel. The parent creates the concurrency for these methods and controls it, waiting for the threads to finish and returning the results (an array of results, one element per child).

The Distributor pattern can also be used in other programs. It was used three times in the PSRS algorithm, and can be applied to any data–parallel problem.

After the design stage, the user selects the Distributor pattern template and instantiates the template. For the Distributor pattern template, the user specifies the names of the concrete classes for the parent and child classes (again, the abstract classes are automatically named) and a list with the following fields:

1. The name of the method that should be invoked concurrently on the child objects.
2. The return type for the child implementation of the method. The parent returns an array of this type, unless the type is `void`.
3. The arguments to the parent implementation of the method. For one–dimensional array parameters, the distribution of the parameter over the child objects must also be specified. Currently supported distributions are pass through, one element per child, striped distribution, block distribution, and neighbour distribution (child i gets a two element array of elements i and $i + 1$ from the original array). All other arguments are passed to the children directly (pass through distribution).

Note the third field of the tuples allows for one–dimensional array arguments to be automatically and correctly distributed to the child objects. For instance, we use the neighbour distribution in the first part of the fourth phase to distribute the partitioned elements to the children of the `Merger Container` class. We also distribute an array of indices, one per child, in the second part of the fourth phase so each child knows where it should merge its sorted partition. The code to distribute these arguments is part of the framework for the pattern and is not written by the user. The last parameter to the Distributor template, the number of children, is specified as a constructor argument in the generated framework.

Using the Framework Based on the specification of the pattern templates for this program, the structure of the framework for the PSRS program is given in Figure 8. The two uses of the Method Sequence framework are the `Sorter Sequence` and `Sorter Facade` class pair, and the `Merger Sequence` and `Merger Facade` pair. The two uses of the Distributor framework are the `Data Container` and `Data Child` pair, and the `Merger Container` and `Merger Child` pair. When generated, the framework does not contain the necessary references for composing the different frameworks; creating these references is covered in Section 3.2.5. However, any references needed in a particular framework are supplied in the generated code. For instance, the abstract sequence class has a reference to the Facade object in the Method Sequence framework. The actual object is supplied by the user to the constructor for the sequence class.

Both the Method Sequence and the Distributor frameworks have different hook methods that can be implemented. The sequence of method calls is shown in Figure 8. For the sequence object in the Method Sequence framework, these hook methods are:

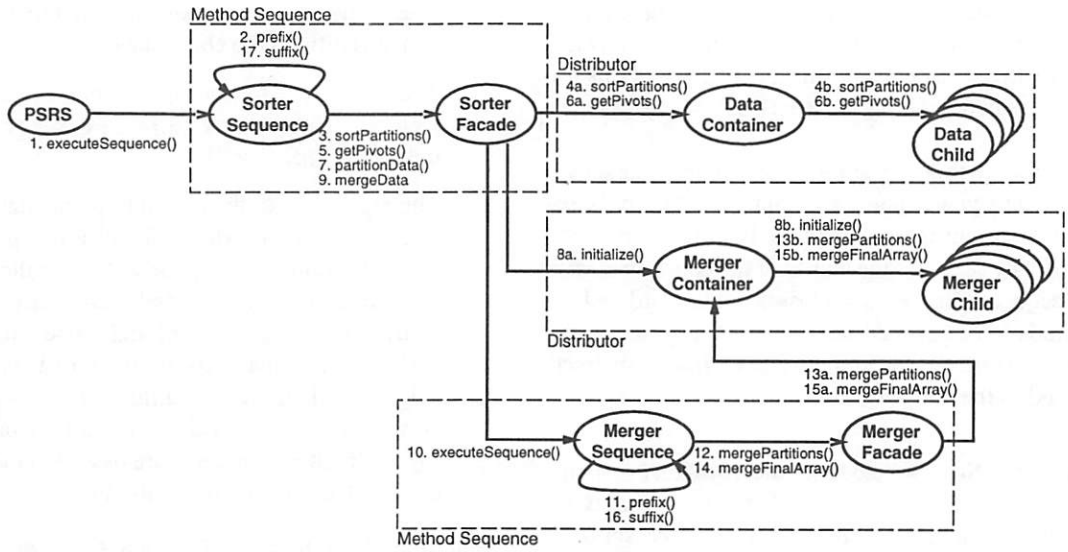


Figure 8: The structure of the PSRS program. The methods are executed in the order in which they are numbered.

`prefix()` This method is invoked before the methods in the sequence are executed. It can be used for any initialization required by either the sequence or Facade objects. For instance, in the `Sorter Sequence` class this method can be used to generate the data to be sorted.

`suffix()` This method is invoked after the methods in the sequence are executed. It can be used for any cleanup or postprocessing required by the program. For instance, in the `Sorter Sequence` class this method may verify the sort.

The hook methods for the Facade object are the sequence methods, the parameterless methods in the list of method names. These methods implement the different phases of the application. A phase has finished when its associated method returns, so all concurrent activity generating results used in another phase must be complete. Any partial results are stored in the Facade object.

For the Distributor framework, the hook methods are the child implementations of the methods specified in the last pattern template parameter. Each child operates independently, without reference to other child objects. These methods can operate on any state that has been distributed across the child objects or can be used to invoke methods on any distributed arguments in parallel. The parent object provides the structural code for this pattern, and has no hook methods. To assist the user, the signatures for the child methods are automatically generated and included in the concrete child class.

3.2.4 Evaluation

Since PSRS is a parallel algorithm, there is no sequential version. Therefore, we chose a sequential quicksort algorithm as a baseline for comparison. The sequential sorting program was 102 lines of Java code, used to initialize the data and verify the sort. The sorting algorithm was the quicksort method from the JGL library [9], which is 255 lines of Java code. The PSRS program, including the framework and collaborator classes, totaled 1252 lines of code (not including the JGL sorting code), 700 of which are user code. 414 lines of the user code are in the three classes `Data Child`, `Merger Child`, and `Data Container`. These classes contain most of the code for the application (the `Data Container` object is the single processor used for the second phase). Of the remaining classes, the two Facade classes and mainline are the largest. However, the methods in these classes consist mainly of accessors and, particularly in the two Facade objects, one line methods for delegating a method. The mainline also interprets command line arguments, and creates the `Sorter Sequence` object and the container for the data to be sorted.

In contrast to the reaction–diffusion example, the PSRS algorithm cannot make much use of the code from the sequential version. The problem is that the best parallel algorithm is not necessarily a parallel version of the best sequential algorithm. For instance, the performance of parallel quicksort peaks at a speedup of 5 to 6 regardless of the number of processes [13]. In these cases, writing the parallel algorithm requires more effort, as we see

with this problem. Nevertheless, the framework supplies about half of the code automatically, including the error–prone concurrency and synchronization code.

The performance results for PSRS, collected using the same environment given in Section 3.1.3, are shown in Table 2. These timings are only for sorting the data. Data initialization and sort verification are not included.

Unlike the reaction–diffusion example, both JIT and non–JIT versions of the PSRS program show good speedups, scaling to 16 processors. The principle reason for this improvement is that there are fewer synchronization points in PSRS; five for the entire program versus two per iteration of the mesh loop. In addition, the PSRS algorithm does more work between synchronization points, even with the smaller data set, reducing the overall cost of synchronization further.

From this example, we can see that CO_2P_3S also supports the development of explicitly parallel algorithms. The principle difficulty in implementing this kind of parallel algorithm is that little sequential code can be used in the parallel program, forcing the user to write more code (as we can see by the amount of user code needed for PSRS). Support for explicitly parallel algorithm development is crucial because a good parallel algorithm is not always derived from the best sequential algorithm.

3.2.5 Composition of Frameworks

Unlike the reaction–diffusion program, the PSRS example used multiple design pattern templates in its implementation, and required the resulting frameworks to be composed into a larger program. We explain briefly how this composition is accomplished, which also provides insights on how the user can augment the generated framework at the Patterns Layer.

In CO_2P_3S frameworks, composition is treated as it is in normal object–oriented programs, by delegating methods to collaborating objects. Note that the framework implementing a design pattern is still a group of objects providing an interface to achieve some task. For instance, in the code in Figure 6, the collector object provides an interface for the user to start the mesh computation and get the results, but the creation and control of the parallelism is hidden in the collector. If another framework has a reference to a collector object, it can use the Mesh framework as it would any other collaborating object, providing framework composition in a way compatible with object–oriented programming.

To compose frameworks in this fashion, the frameworks must be able to obtain references to other collaborating frameworks. This can be done in three ways: passing the references as arguments to a method (normal or hook) and caching the reference, instantiating the collaborating framework in the framework that requires it (in a method or constructor), or augmenting constructors with new arguments. The first two ways are fairly straightforward. The second method of obtaining a reference is used in the third phase of PSRS since the `Merger Container` object cannot be created until the pivots have been obtained from the second phase.

The third method of obtaining a reference, augmenting the constructor for a framework, requires more discussion as it is not always possible. We should first note that this option is open to users because the CO_2P_3S system requires the user to create some or all of the objects that make up a given framework (as shown in Figure 6). In general, users can augment the constructors of any objects they are responsible for creating. For the Mesh framework, the user can augment the constructor for the collector object. However, the added state can only be used to influence the parallel execution of a framework at the Patterns Layer if the class with the augmented constructor also has hook methods the user can implement. Otherwise, the user has no entry point to the structural code and the additional state cannot be used in the parallel portion of that framework. For instance, the user can augment the constructor for the parent object in a Distributor framework, but since the parent has no hook methods this state cannot influence the parallel behaviour of that object. However, new state can always be used in any additional sequential methods implemented in the framework.

4 Related Work

We examine work related to the pattern, pattern template, and framework aspects of the CO_2P_3S system.

Patterns There are too many concurrent design patterns to list them all. Two notable sources of these patterns are the ACE framework [11] and the concurrent design pattern book by Lea [6]. This work provides more patterns and attempts to provide a development system for pattern–based parallel programming. Specifically, our pattern templates and generated frameworks automate the use of a set of supported patterns.

Pattern Templates There are many graphical parallel programming systems, such as Enterprise [10, 14], DP-

Number of Elements	JIT disabled				JIT enabled			
	2 proc.	4 proc.	8 proc.	16 proc.	2 proc.	4 proc.	8 proc.	16 proc.
12,500,000	1.76 1519 sec	3.43 777 sec	6.74 395 sec	12.02 222 sec	1.73 567 sec	3.65 269 sec	7.09 138 sec	12.65 78 sec

Table 2: Speedups for the PSRS example. Wall clock times, rounded to the second, are also provided.

nDP [15, 14], Mentat [5], and HeNCE [2]. Enterprise provides a fixed set of templates (called *assets*) for the user, but requires the user to write code to correctly implement the chosen template, without checking for compliance. Further, applications are written in C, not an object–oriented language. Mentat and HeNCE do not use pattern templates, but rather depict programs visually as directed graphs, compromising correctness. DPnDP is similar to Mentat except that nodes in the graph may contain instances of design patterns communicating using explicit message passing. In addition, the system provides a method for adding new templates to the tool.

P^3L [1] provides a language solution to pattern–based programming, providing a set of design patterns that can be composed to create larger programs. Communication is explicit and type–checked at compile–time. However, new languages impose a steep learning curve on new users. Also, the language is not object–oriented.

Frameworks Sefika *et al.* [12] have proposed a model for verifying that a program adheres to a specified design pattern based on a combination of static and dynamic program information. They also suggest the use of run–time assertions to ensure compliance. In contrast, we ensure adherence by generating the code for a pattern. We do not include assertions because we allow users to modify the generated frameworks at lower levels of abstraction. These modifications can be made to increase performance or to introduce a variation of a pattern template that is not available at the Patterns Layer.

In addition to verifying programs, Sefika *et al.* also suggest generating code for a pattern. Budinsky *et al.* [3] have implemented a Web–based system for generating code implementing the patterns from Gamma *et al.* [4]. The user downloads the code and modifies it for its intended application. Our system generates code that allows the user the opportunity to introduce application–specific functionality without allowing the structure of the framework to be modified until the performance–tuning stage of development. This allows us to enforce the parallel constraints of the selected pattern template.

Each of the PPSs mentioned above differ with respect to openness. Enterprise, Mentat, HeNCE, and P^3L fail to provide low–level performance tuning. However, Enterprise provides a complete set of development and debugging tools in its environment. DPnDP provides performance tuning capabilities by allowing the programmer to use the low–level libraries used in its implementation. Instead, we provide multiple abstractions for performance tuning, providing the low–level libraries only at the lowest level of abstraction.

5 Conclusions and Future Work

This paper presented some of the parallel design patterns and associated frameworks supported by the CO_2P_3S parallel programming system. We demonstrated the utility of these patterns and frameworks at the first layer of the CO_2P_3S programming model by showing how two applications, reaction–diffusion texture generation and parallel sorting by regular sampling, can be implemented. Further, we have shown that our frameworks can provide performance benefits.

We also introduced the concept of phased design patterns to express temporal relationships in parallel programs. These relationships may determine when to use parallelism and how that parallelism, if used, should be implemented. These phased patterns recognize that not every operation on an object has sufficient granularity to be run in parallel, and that a single parallel design pattern is often insufficient to efficiently parallelize an entire application. Instead, the parallel requirements of an application change as the application progresses. Phased patterns provide a mechanism for expressing this change.

Currently, we are prototyping the CO_2P_3S system in Java. We are also looking for other parallel design patterns that can be included in the system, such as divide–and–conquer and tree searching patterns. Once the programming system is complete, we will investigate allowing users to add support for their own parallel design patterns by including new pattern templates and frameworks in CO_2P_3S (as can be done in DPnDP [15]), creating a tool set to assist with debugging and tuning programs (such as the Enterprise environment [10]), and conducting usability studies [14, 16].

Acknowledgements

This research was supported by grants from the National Science and Engineering Research Council of Canada. We are indebted to Doug Lea for numerous comments and suggestions that significantly improved this paper.

References

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[2] A. Beguelin, J. Dongarra, A. Giest, R. Manchek, and K. Moore. HeNCE: A heterogeneous network computing environment. Technical Report UT-CS-93-205, University of Tennessee, August 1993.

[3] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[5] A. Grimshaw. Easy to use object–oriented parallel programming with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[6] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, 1997.

[7] S. MacDonald. Parallel object–oriented pattern catalogue. Available at http://www.cs.ualberta.ca/~stevem/publications.html, 1998.

[8] S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 267–274. Springer–Verlag, 1997.

[9] ObjectSpace, Inc. *ObjectSpace JGL: The Generic Collection Library for Java, Version 3.0*, 1997. http://www.objectspace.com.

[10] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[11] D. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *Proceedings of the 12th Sun Users Group Conference*, 1994. A list of the individual patterns can be found at http://siesta.cs.wustl.edu/~schmidt/patterns-ace.html.

[12] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high–level design models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 387–396, 1996.

[13] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[14] A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

[15] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 230–240, 1996.

[16] G. Wilson and H. Bal. Using the Cowichan problems to assess the usability of Orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

[17] A. Witkin and M. Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proccedings)*, 25(4):299–308, July 1991.

Intercepting and Instrumenting COM Applications

Galen C. Hunt
Microsoft Research
One Microsoft Way
Redmond, WA 98052
galenh@microsoft.com

Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627
scott@cs.rochester.edu

Abstract

Binary standard object models, such as Microsoft's Component Object Model (COM) enable the development of not just reusable components, but also an incredible variety of useful component services through run-time interception of binary standard interfaces. Interception of binary components can be used for conformance testing, debugging, profiling, transaction management, serialization and locking, cross-standard middleware interoperability, automatic distributed partitioning, security enforcement, clustering, just-in-time activation, and transparent component aggregation.

We describe the implementation of an interception and instrumentation system tested on over 300 COM binary components, 700 unique COM interfaces, 2 million lines of code, and on 3 major commercial-grade applications including Microsoft PhotoDraw 2000. The described system serves as the foundation for the Coign Automatic Distributed Partitioning System (ADPS), the first ADPS to automatically partition and distribute binary applications.

While the techniques described in this paper were developed specifically for COM, they have relevance to other object models with binary standards, such as individual CORBA implementations.

1. Introduction

Widespread adoption of Microsoft's Component Object Model (COM) [16, 25] standard has produced an explosion in the availability of binary components, reusable pieces of software in binary form. It can be argued that this popularity is driven largely by COM's binary standard for component interoperability.

While binary compatibility is a great boon to the market for commercial components, it also enables a wide range of unique component services through interception. Because the interfaces between COM components are well defined by the binary standard, a component service can exploit the binary standard to intercept inter-component communication and interpose itself between components.

Interception of binary components can be used for conformance testing, debugging, distributed communication, profiling, transaction management, serialization and locking, cross-standard middleware interoperability, automatic distributed partitioning, security enforcement, clustering and replication, just-in-time activation, and transparent component aggregation.

In this paper, we describe an interception system proven on over 300 COM binary components, 700 unique COM interfaces, and 2 million lines of code [5]. We have extensively tested our COM interception system on three major commercial-grade applications: the MSDN Corporate Benefits Sample [12], Microsoft PhotoDraw 2000 [15], and the Octarine word-processor from the Microsoft Research COM Applications Group. The interception system serves as the foundation for the Coign Automatic Distributed Partitioning System (ADPS) [7] [8], the first ADPS to automatically partition and distribute binary applications.

In the next section, we describe the fundamental features of COM as they relate to the interception and instrumentation of COM applications. Sections 3 and 4 explain and evaluate our mechanisms for intercepting object instantiation requests and inter-object communication respectively. We describe related work in Section 5. In Section 6, we present our conclusions and propose future work.

2. COM Fundamentals

COM is a standard for creating and connecting components. A COM component is the binary template from which a COM object is instantiated. Due to COM's binary standard, programmers can easily build applications from components, even components for which they have no source code. COM's major features include multiple interfaces per object, mappings for common programming languages, standard-mandated binary compatibility, and location-transparent invocation.

2.1. Polymorphic Interfaces

All first-class communication in COM takes place through interfaces. An interface is a strongly typed reference to a collection of semantically related functions. An interface is identified by a 128-bit globally unique identifier (GUID). An explicit agreement between two components to communicate through a

named interface contains an implicit contract of the binary representation of the interface.

Microsoft Interface Definition Language (MIDL)

Figure 1 contains the definitions of two interfaces: `IUnknown` and `Istream` in the *Microsoft Interface Definition Language* (MIDL). Syntactically, MIDL is very similar to C++. To clarify the semantic features of interfaces, MIDL attributes (enclosed in square brackets []) can be attached to any interface, member function, or parameter. Attributes specify features such as the data-flow direction of function arguments, the size of dynamic arrays, and the scope of pointers. For example, the `[in, size_is(cb)]` attribute on the `pb` argument of the `Write` function in Figure 1 declares that `pb` is an input array with `cb` elements.

```
[uuid(00000000-0000-0000-C000-000000000046)]
interface IUnknown
{
    HRESULT QueryInterface(
        [in] REFIID riid,
        [out,iid_is(riid)] void **ppObj);
    ULONG AddRef();
    ULONG Release();
};

[uuid(b3c11b80-9e7e-11d1-b6a5-006097b010e3)]
interface IStream : IUnknown
{
    HRESULT Seek(
        [in] LONG nPos);
    HRESULT Read(
        [out,size_is(cb)] BYTE *pb,
        [in] LONG cb);
    HRESULT Write(
        [in,size_is(cb)] BYTE *pb,
        [in] LONG cb);
};
```

Figure 1. MIDL for Two Interfaces.

The MIDL definition of an interface describes its member functions and their parameters in sufficient detail to support location-transparent invocation.

IUnknown

The `IUnknown` interface, listed in Figure 1, is special. All COM objects must support `IUnknown`. Each COM interface must include the three member functions from `IUnknown`, namely: `QueryInterface`, `AddRef`, and `Release`. `AddRef` and `Release` are reference-counting functions for lifetime management. When an object's reference count goes to zero, the object is responsible for freeing itself from memory.

COM objects can support multiple interfaces. Clients dynamically bind to a new interface by calling `QueryInterface`. `QueryInterface` takes as input the GUID of the interface to which the client would like to bind and returns a pointer to the new interface. Through run-time invocation of `QueryInterface`, clients can determine the exact functionality supported by any object.

2.2. Common Language Mappings

The MIDL compiler maps interface definitions into formats usable by common programming languages. Figure 2 contains the C++ abstract classes generated by the MIDL compiler, for the interfaces in Figure 1. MIDL has straightforward mappings into other compiled languages such as C and Java. In addition, the MIDL compiler can store metadata in binary files called *type libraries*. Many development tools can import type libraries. Type libraries are well suited for scripting languages such as the Visual Basic Scripting Edition in Internet Explorer [11].

```
class IUnknown
{
  public:
    virtual HRESULT QueryInterface(
                        REFIID riid,
                        void **ppObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};

class IStream : IUnknown
{
  public:
    virtual HRESULT Seek(
                        LONG nPos) = 0;
    virtual HRESULT Read(
                        BYTE *pb,
                        LONG cb) = 0;
    virtual HRESULT Write(
                        BYTE *pb,
                        LONG cb) = 0;
};
```

Figure 2. C++ Language Mapping.

The MIDL compiler maps a COM interface into an abstract C++ class.

2.3. Binary Compatibility

In addition to language mappings, COM specifies a platform-standard binary mapping for interfaces. The binary format for a COM interface is similar to the common format of a C++ *virtual function table* (VTBL, pronounced "V-Table"). All references to interfaces are stored as interface pointers (an indirect pointer to a virtual function table). Figure 3 shows the binary mapping of the `IStream` interface.

Each object is responsible for allocating and releasing the memory occupied by its interfaces. Quite often, objects place per-instance interface data immediately following the interface virtual-function-table pointer. With the exception of the virtual function table and the pointer to the virtual function table, the object memory area is opaque to the client.

The standardized binary mapping enforces COM's language neutrality. Any language that can call a function through a pointer can use COM objects. Any language that can export a function pointer can create COM objects.

COM components are distributed either in application executables (.EXE files) or in *dynamic link libraries* (DLLs).

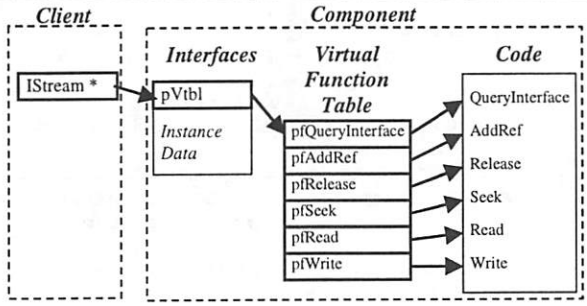


Figure 3. Binary Interface Mapping.

COM defines a standard binary mapping for interfaces. The format is similar to the common representation of a C++ pure abstract virtual function table.

2.4. Location Transparency

Binary compatibility is important because it facilitates true location transparency. A client can communicate with a COM object in the same process (*in-process*), in a different process (*cross-process*), or on an entirely different machine (*cross-machine*). The location of the COM object is completely transparent to both client and component because in each case invocation takes place through an interface's virtual function table.

Interface Proxies and Stubs

Location transparency is achieved through proxies and stubs generated by the MIDL compiler. *Proxies* marshal function arguments into a single message that can be transported between address spaces or between machines. *Stubs* unmarshal messages into function calls. Interface proxies and stubs copy data structures with deep-copy semantics. In theory, proxies and stubs come in pairs—the first for marshaling and the second for unmarshaling. In practice, COM generally combines code for the proxy and stub for a specific interface into a single reusable binary. COM proxies and stubs are similar in purpose to CORBA [19, 23] stubs and skeletons. However, their implementations vary because COM proxies and stubs are only used when inter-object communication crosses process boundaries.

In-Process Communication

For best performance, components reside in the client's address space. An application invokes an in-process object directly through the interface virtual function table. In-process communication has the same cost as a C++ virtual function call because it uses neither interface proxies nor stubs. The primary drawback of in-process objects is that they share the same protection domain as the application. The application cannot protect itself from erroneous or malicious resource access by the object.

Cross-Process Communication

To provide the application with security, objects can be located in another operating-system process. The application communicates with cross-process objects through interface proxies and stubs. The application invokes the object through an indirect call on an interface virtual function table. In this case, however, the virtual function table belongs to the interface proxy. The proxy marshals function arguments into a buffer and transfers execution to the object's address space where the interface stub unmarshals the arguments and calls the object through the interface virtual function table in the target address space. Marshaling and unmarshaling are completely transparent to both application and component.

Cross-Machine Communication

Invocation of distributed objects is very similar to invocation of cross-process objects. Cross-machine communication uses the same interface proxies and stubs as cross-process communication. The primary difference is that once the function arguments have been marshaled, COM sends the serialized message to the destination machine using the DCOM protocol [3], a superset of the Open Group's Distributed Computing Environment Remote Procedure Call (DCE RPC) protocol [4].

3. Interception of Object Instantiations

COM objects are dynamic objects. Instantiated during an application's execution, objects communicate with the application and each other through dynamically bound interfaces. An object frees itself from memory after all references to it have been released by the application and other objects.

Applications instantiate COM objects by calling API functions exported from a user-mode COM DLL. Ap-

plications bind to the COM DLL either statically or dynamically.

Static binding to a DLL is very similar to the use of shared libraries in most UNIX systems. Static binding is performed in two stages. At link time, the linker embeds in the application binary the name of the DLL, a list of all imported functions, and an indirect jump table with one entry per imported function. At load time, the loader maps all imported DLLs into the application's address space and patches the indirect jump table entries to point to the correct entry points in the DLL image.

Dynamic binding occurs entirely at run time. A DLL is loaded into the application's address space by calling the `LoadLibrary` Win32 function. After loading, the application looks for procedures within the DLL using the `GetProcAddress` function. In contrast to static binding, in which all calls use an indirect jump table, `GetProcAddress` returns a direct pointer to the entry point of the named function.

BindMoniker	OleCreateDefaultHandler
CoCreateInstance	OleCreateEx
CoCreateInstanceEx	OleCreateFontIndirect
CoGetClassObject	OleCreateFromData*
CoGetInstanceFromFile	OleCreateFromFile*
CoRegisterClassObject	OleCreateLink*
CreateAntiMoniker	OleCreateStaticFromData
CreateBindCtx	OleGetClipboard
CreateClassMoniker	OleLoad
CreateDataAdviseHolder	OleLoadFromStream
CreateFileMoniker	OleLoadPicture
CreateGenericComposite	OleLoadPictureFile
CreateItemMoniker	OleRegEnumFormatEtc
CreateOleAdviseHolder	OleRegEnumVerbs
CreatePointerMoniker	StgCreateDocfile
GetRunningObjectTable	StgCreateDocfileOn*
MkParseDisplayName	StgGetIFillLockBytesOn*
MonikerCommonPrefixWith	StgOpenAsyncDocfileOn*
MonikerRelativePathTo	StgOpenStorage
OleCreate	StgOpenStorageOn*

Figure 4. Object Instantiation Functions.

COM supports approximately 50 functions capable of creating instantiation a new object. However, most instantiations request use either `CoCreateInstance` or `CoCreateInstanceEx`.

The COM DLL exports approximately 50 functions capable of instantiating new objects; these are listed in Figure 4. With few exceptions, applications instantiate objects exclusively through the `CoCreateInstance` function or its successor, `CoCreateInstanceEx`. From the instrumentation perspective there is little difference among the COM API functions. For brevity, we use `CoCreate` as a placeholder for any function that instantiates new COM objects.

3.1. Alternatives for Instantiation Interception

To intercept all object instantiations, instrumentation should be called at the entry and exit of each object instantiation function.

Figure 5 enumerates the techniques available for intercepting functions; namely: source-code call replacement, binary call replacement, DLL redirection, DLL replacement, breakpoint trapping, and inline redirection.

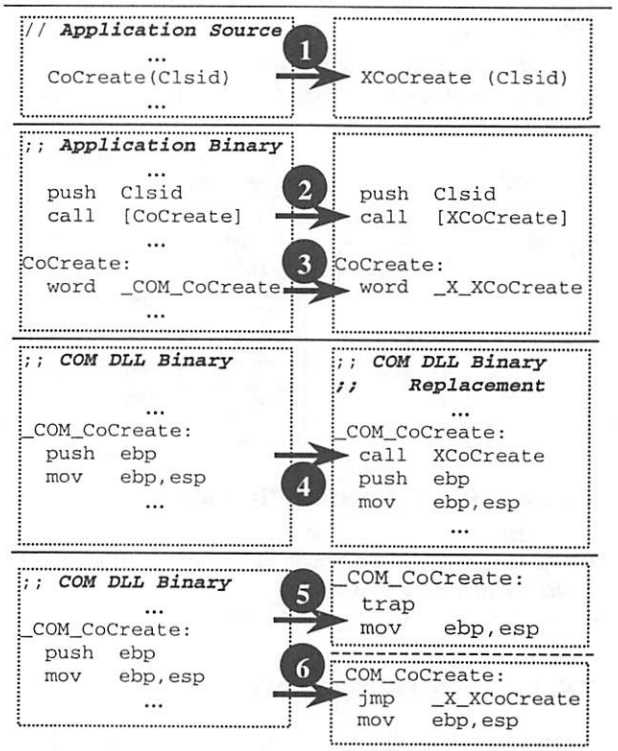


Figure 5. Intercepting Instantiation Calls.

Object instantiation calls can be intercepted by 1) call replacement in the application source code; 2) call replacement in the application binary; 3) DLL redirection; 4) DLL replacement; 5) trapping in the COM DLL; and 6) inline redirection in the COM DLL.

Call replacement in application source code.

Calls to the COM instantiation functions can be replaced with calls to the instrumentation by modifying application source code. The major drawback of this technique is that it requires access to application source code.

Call replacement in application binary code.

Calls to the COM instantiation functions can be replaced with calls to the instrumentation by modifying application binaries. While this technique does not require source code, replacement in the application binary does require the ability to identify all applicable

call sites. To facilitate identification of all call sites, the application must be linked with substantial symbolic information.

DLL redirection.

The import entries for COM APIs in the application can be modified to point to another library. Redirection to another DLL can be achieved either by replacing the name of the COM DLL in the import table before load time or by replacing the function addresses in the indirect jump table after load. Unfortunately, redirecting to another DLL through either of the import tables fails to intercept dynamic calls using `LoadLibrary` and `GetProcAddress`.

DLL replacement.

The only way to guarantee interception of a specific DLL function is to insert the interception mechanism into the function code. The most obvious method is to replace the COM DLL with a new version containing instrumentation. DLL replacement requires source access to the COM DLL library. It also unnecessarily penalizes all applications using the COM DLL, whether they use the additional functionality or not.

Breakpoint trapping of the COM DLL.

Rather than replace the DLL, the interception mechanism can be inserted into the image of the COM DLL after it has been loaded into the application address space. At run time, the instrumentation system can insert a breakpoint trap at the start of each target instantiation function. When execution reaches the function entry point, a debugging exception is thrown by the trap and caught by the instrumentation system. The major drawback to breakpoint trapping is that debugging exceptions suspend all application threads. In addition, the debug exception must be caught in a second operating-system process. Interception via breakpoint trapping has a high performance penalty.

Inline redirection of the COM DLL.

The most favorable method for intercepting DLL functions is to inline the redirection call. At load time, the first few instructions of the target instantiation function are replaced with a jump instruction to a detour function in the instrumentation. Replacing the first few instructions is usually a trivial operation as these instructions are normally part of the function prolog generated by a compiler and not the targets of any branches. The replaced instructions are used to create a trampoline. When the modified target function is invoked, the jump instruction transfers execution to the detour function in the instrumentation. The detour function passes control to the remainder of the target function by invoking the trampoline.

3.2. Evaluation of Instantiation Interception

Our instrumentation system uses inline indirection to intercept object instantiation calls. At load time, our instrumentation replaces the first few instructions of the target function with a jump to the instrumentation detour function. Pages for code sections are mapped into a processes' address space using copy-on-write semantics. Calls to `VirtualProtect` and `FlushInstructionCache` enable modification of code pages at run time. Instructions removed from the target function are placed in a statically allocated trampoline routine. As shown in Figure 6, the trampoline allows the detour function to invoke the target function without interception.

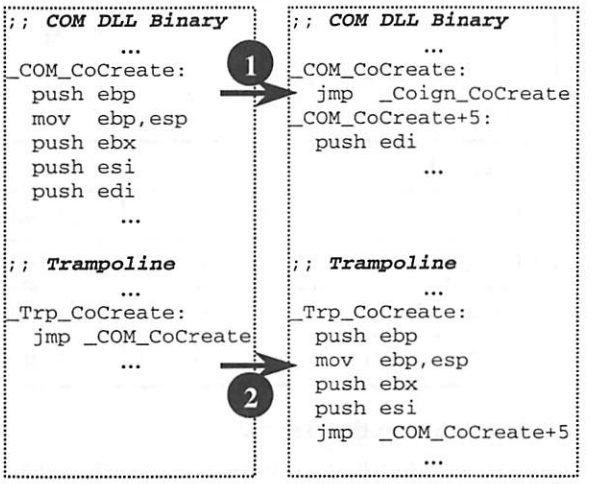


Figure 6. Inline Redirection.

The first few instructions of the target API function are moved to the trampoline and replaced with a jump to the interception system. The trampoline effectively invokes the API function without interception. On the Intel x86 architecture, a jump instruction occupies five bytes.

Although inline indirection is complicated by the variable-length instruction set of the Intel x86 architecture, its low run-time cost and versatility more than offset the development penalty. Inline redirection of the `CoCreateInstance` function has less than a 3% overhead, which is more than an order of magnitude smaller than the penalty for breakpoint trapping. Table 1 lists the average invocation time of the target function within a loop consisting of 10,000 iterations. The invocation times include the cost of redirection, but not any additional instrumentation. Unlike DLL redirection, inline redirection correctly intercepts both statically and dynamically bound invocations. Finally, inline redirection is much more flexible than DLL redirection or application code modification. Inline redirection of any API function can be selectively enabled for each proc-

ess individually at load time based on the needs of the instrumentation.

To apply inline redirection, our instrumentation system must be loaded into the application's address space before the application executes. The current system is packaged as a DLL and post-linked to the application binary with a binary rewriter. Once loaded into the application address space, instrumentation is inlined into system DLL images. Mechanisms for inserting the interception system into an application's address space are described fully in a paper on our Detours package [6].

Function vs. Interception Technique	Empty Function	CoCreateInstance
Direct Call	0.11us	14.84us
DLL Redirection	0.14us	15.19us
DLL Replacement	0.14us	15.19us
Breakpoint Trap	229.56us	265.85us
Inline Redirection	0.15us	15.19us

Table 1. Interception Times.

Listed are the times for intercepting either an empty function or `CoCreateInstance` on a 200MHz Pentium PC.

4. Intercepting Inter-Object Calls

The bulk the interception system's functionality is devoted to identifying interfaces, understanding their relationships to each other, and quantifying the communication through them. This section describes how our system intercepts interface calls.

Invoking an interface member function is similar to invoking a C++ member function. The first argument to any interface member function is the "`this`" pointer, the pointer to the interface. Figure 7 lists the C++ and C syntax to invoke an interface member function.

4.1. Alternatives for Invocation Interception

There are four techniques, described below, available to intercept member function invocations:

Replace the interface pointer.

Rather than return the object's interface pointer, the interception system can return a pointer to an interface of its own making. When the client attempts to invoke an interface member function, it will invoke the instrumentation, not the object. After taking appropriate steps, the instrumentation "forwards" the request to the object by directly invoking the object interface. In one sense, replacing the interface pointer is functionally similar to using remote interface proxies and stubs. For remote marshaling, COM replaces a remote interface pointer with a local interface pointer to an interface proxy.

Replace the interface virtual function table pointer.

The runtime can replace the virtual function table pointer in the interface with a pointer to an instrumentation-supplied virtual function table. The instrumentation can forward the invocation to the object by keeping a private copy of the original virtual function table pointer.

Replace function pointers in the interface virtual function table.

Rather than intercept the entire interface as a whole, the interception system can replace each function pointer in the virtual function table individually.

Intercept object code.

Finally, the instrumentation system can intercept member-function calls at the actual entry point of the function using inline redirection.

```
IStream *pIStream;

// C++ Syntax
pIStream->Seek(nPos);

// C Syntax
pIStream->pVtbl->pfSeek(pIStream, nPos);
```

Figure 7. Invoking an Interface Function.

Clients invoke interface member functions through the interface pointer. The first parameter to the function (hidden in C++) is the "`this`" pointer to the interface.

4.2. COM Programming Idioms

The choice of an appropriate technique for intercepting member functions is constrained by COM's binary standard for object interoperability and common COM programming idioms. Our interception system attempts to deduce the identity of the each called object, the static type of the called interface, the identity of the called member function, and the static types of all function parameters. In addition, our interception degrades gracefully. Even if not all of the needed information

can be determined, the interception system continues to function correctly.

By design, the COM binary standard restricts the implementation of interfaces and objects to the degree necessary to insure interoperability. COM places four specific restrictions on interface design to insure object interoperability. First, a client accesses an object through its interface pointers. Second, the first item pointed to by an interface pointer must be a pointer to a virtual function table. Third, the first three entries of the virtual function table must point to the `QueryInterface`, `AddRef` and `Release` functions for the interface. Finally, if a client intends to use an interface, it must insure that the interface's reference count has been incremented.

As long as an object programmer obeys the four rules of the COM binary standard, he or she is completely free to make any other implementation choices. For example, the component programmer is free to choose any appropriate memory layout for object and per-instance interface data. This lack of implementation constraint is not an accident. The original designers of COM were convinced that no one implementation (even of something as universal as the `QueryInterface` function) would be suitable for all users. Instead, they attempted to create a specification that enabled binary interoperability while preserving all other degrees of freedom.

Specification freedom breeds implementation diversity. This diversity is manifest in the number of common programming idioms employed by COM component developers. These idioms are described here in sufficient detail to highlight the constraints they place on the implementation of a COM interception and instrumentation system. Each of these idioms has broken at least one other COM interception system or preliminary versions of our interception system.

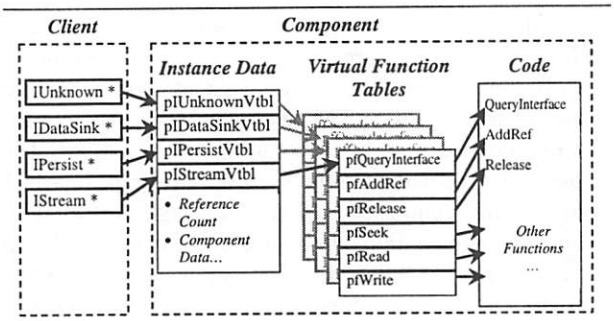


Figure 8. Simple Object Layout.

The object instance is allocated as a single memory block. The block contains one VTBL pointer for each supported interface, an instance reference count, and other object-specific data. All interfaces share common implementations of `QueryInterface`, `AddRef`, and `Release`.

Simple Multiple-Interface Objects

Most objects support at most roughly a dozen interfaces with no duplicates. It is common practice to lay out these simple objects in a memory block containing one VTBL pointer per interface, a reference count, and internal object variables; see Figure 8. Within the object's member functions, a constant value is added to the "`this`" pointer to find the start of the memory block and to access object variables. All of the object interfaces use a common pair of `AddRef` and `Release` functions to maintain the object reference count.

Multiple-Instance and Tear-off Interfaces

Sometimes, an object must support multiple copies of a single interface. *Multiple-instance* interfaces are often used for iteration. A new instance of the interface is allocated for each client. Multiple-instance interfaces are typically implemented using a *tear-off* interface. A tear-off interface is allocated as a separate memory block. The tear-off interface contains the interface's VTBL pointer, an interface-specific reference count, a pointer to the object's primary memory block, and any instance-specific data. In addition to multiple-instance interfaces, tear-off interfaces are often used to implement rarely accessed interfaces when object memory size must be minimized, (i.e. when the cost of the extra four bytes for a VTBL pointer per object instance is too expensive).

Universal Delegators

Objects commonly use a technique called *delegation* to export interfaces from another object to a client. Delegation is often used when one object aggregates services from several other objects into a single entity. The aggregating object exports its own interfaces, which delegate their implementation to the aggregated objects. The delegating interface calls the aggregated interface. This implementation is interface specific, code intensive, and requires an extra procedure call during invocation. The implementation is code intensive because delegating code must be written for each interface type. The extra procedure call becomes particularly important if the member function has a large number of arguments or multiple delegators are nested through layers of aggregation.

An obvious optimization and generalization of delegation is the *universal delegator*. A universal delegator is a type-independent, re-usable delegator. The data structure for a universal delegator consists of a VTBL pointer, a reference count, a pointer to the aggregated interface, and a pointer to the aggregating object. Upon invocation, a member function in the universal delegator replaces the "`this`" pointer on the argument stack with the pointer to the delegated interface and jumps directly to the entry point of the appropriate member

function in the aggregated interface. The universal delegator is "universal" because its member functions need know nothing about the type of interface to which they are delegating; they reuse the invoking call frame. Implemented in a manner similar to tear-off interfaces, universal delegators are instantiated on demand, one per delegated interface with a common VTBL shared among all instances.

Explicit VTBL Pointer Comparison.

Rather than using explicit constant offsets, some COM components implemented in C locate the start of an object's main memory block by comparing VTBL interface pointers. For example, the `IStream::Seek` member function of the object in Figure 8 starts with its "this" pointer pointing to `pIStreamVtbl`. The object locates the start of its memory structure by decrementing the "`this`" pointer until it points to a VTBL pointer equal to the known location of the VTBL for `IUnknown`. This calculation will produce erroneous results if an interception system has replaced the VTBL pointer.

Explicit Function Pointer Comparison.

In a manner similar to VTBL pointer comparison, some components perform calculations assuming that function pointers in the VTBL will have known values. These calculations break if the interception system has replaced a VTBL function pointer.

4.3. Interface Wrapping

Our instrumentation system intercepts invocation of interface member functions by replacing the interface pointer given to the object's client with an interface pointer to a specialized universal delegator, the *interface wrapper*. The implementation of interface wrappers was chosen after evaluating the functionality of possible alternatives and testing their performance against a suite of object-based applications.

For brevity, we often refer to the process of creating an individual interface wrapper and replacing the interface pointer with a pointer to an interface wrapper as *wrapping* the interface. We also refer to interfaces as being *wrapped* or *unwrapped*. A wrapped interface is one to which clients receive a pointer to the interface wrapper. An unwrapped interface is one either without a wrapper or with the interface wrapper removed to yield the original object interface.

Interface wrapping provides an easy way to identify an interface and a ready location to store information about the interface: in the per-instance interface wrapper. Unlike interface wrapping, inline redirection must store per-instance data in an external dictionary. Access to the instance-data dictionary is made difficult because member functions are often re-used by multiple interfaces of dissimilar type. This is definitely the case for universal delegation, but common even for less exotic coding techniques. As a rule, almost all objects reuse the same implementation of `QueryInterface`, `AddRef`, and `Release` for multiple interfaces.

Interface wrapping is robust, does not break application code, and is extremely efficient. Finally, as we shall see in the next section, interface wrapping is central to correctly identifying the object that owns an interface.

4.4. The Interface Ownership Problem

In addition to intercepting interface calls, the interception system attempts to identify which object owns an interface. A major breakthrough in the development of our interception system was the discovery of heuristics to find an interface's owning object.

The interface ownership problem is complicated because to COM, to the application, and to other objects, an object is visible only as a loosely coupled set of interfaces. The object can be identified only through one of its interfaces; it has no explicit object identity.

COM supports the concept of an object identity through the `IUnknown` interface. As mentioned in Chapter 2, every interface must inherent from and implement the three member functions of `IUnknown`, namely: `QueryInterface`, `AddRef`, and `Release`. Through the `QueryInterface` function, a client can query for any interface supported by the object. Every object must support the `IUnknown` interface. An object's `IUnknown` interface pointer *is* the object's *COM identity*. The COM specification states that a client calling `QueryInterface(IID_IUnknown)` on any interface must always receive back the same `IUnknown` interface pointer (the same COM identity).

Unfortunately, an object need not provide the same COM identity (the same `IUnknown` interface pointer) to different clients. An object that exports one COM identity to one client and another COM identity to a second client is said to have a *split identity*. Split identities are especially common in applications in which objects are composed together through a technique known as aggregation. In aggregation, multiple objects operate as a single unit by exporting a common `QueryInterface` function to all clients. Due to split identities, COM objects have no system-wide, unique identifier.

The Obvious Solution

A client can query an interface for its owning `IUnknown` interface (its COM identity). In the most obvious implementation, the interception system could maintain a list of known COM identities for each ob-

ject. The runtime could identify the owning object by querying an interface for its COM identity and comparing it to a dictionary of known identities.

In practice, calling `QueryInterface` to identify the owning object fails because `QueryInterface` is not free of side effects. `QueryInterface` increments the reference count of the returned interface. Calling `Release` on the returned interface would decrement its reference count. However, the `Release` function also has side effects. `Release` instructs the object to check if its reference count has gone to zero and to free itself from memory in the affirmative. There are a few identification scenarios under which the object's reference count does in fact go to zero. In the worse case scenario, attempting to identify an interface's owner would produce the unwanted side effect of instructing the object to remove itself from memory!

Sources of Interface Pointers

To find a correct solution to the interface ownership problem, one must understand how a client receives an interface pointer. It is also important to understand what information is available about the interface. A client can receive an object interface pointer in one of four ways: from one of the COM API object instantiation functions; by calling `QueryInterface` on an interface to which it already holds a pointer; as an output parameter from one of the member functions of an interface to which it already holds a pointer; or as an input parameter on one of its own member functions. Recall that our system intercepts all COM API functions for object instantiation. At the time of instantiation, the interception system wraps the interface and returns to the caller a pointer to the interface wrapper.

An Analogy for the Interface Ownership Problem

The following analogy is helpful for understanding the interface ownership problem. A person finds herself in a large multi-dimensional building. The building is divided into many rooms with doors leading from one room to another. The person is assigned the task of identifying all of the rooms in the building and determining which doors lead to which rooms. Unfortunately, all of the walls in the building are invisible. Additionally, from time to time new doors are added to the building and old doors are removed from the building.

Mapping the analogy to the interface ownership problem; the building is the application, the rooms are the objects, and the doors are the interfaces.

We describe the solution first in terms of the invisible room analogy, then as it applies to the interface ownership problem. In the analogy, the solution is to assign each room a different color and to paint the doors of that room as they are discovered. The person starts her search in one room. She assigns the room a color—say red. Feeling her way around the room, she paints one side of any door she can find without leaving the room. The door must belong to the room because she didn't pass through a door to get to it. After painting all of the doors, she passes through one of the doors into a new room. She assigns the new room a color—say blue. She repeats the door-painting algorithm for all doors in the blue room. She then passes through one of the doors and begins the process again. The person repeats the process, passing from one room to another.

If at some point the person finds that she has passed into a room where the door is already colored, then she knows the identity of the room (by the color on the door). She looks for any new doors in the room, paints them the appropriate color, and finally leaves through one of the doors to continue her search.

The Solution to the Interface Ownership Problem

From the analogy, the solution to the interface ownership problem is quite simple. Each object is assigned a unique identifier. Each thread holds in a temporary variable the identity of the object in which it is currently executing. Any newly found interfaces are instrumented with an interface wrapper. The current object identity is recorded in the interface wrapper as the owning object. Finding the doors in a room is analogous to examining interface pointers passed as parameters to member functions. When execution exits an object, any unwrapped interface pointers passed as parameters are wrapped and given the identity of their originating object. By induction, if an interface pointer is not already wrapped, then it must belong to the current object.

The most important invariant for solving the interface ownership problem is that at any time the interception system must know exactly which object is executing. Stored in a thread-local variable, the current object identifier is updated as execution crosses through interface wrappers. The new object identifier is pushed onto a local stack on entry to an interface. On exit from an interface wrapper (after executing the object's code), the object identifier is popped from the top of the stack. At any time, the interception system can examine the top values of the identifier stack to determine the identity of the current object and any calling objects.

There is one minor caveat in implementing the solution to the interface ownership problem. While clients should only have access to interfaces through interface wrappers, an object should never see an interface wrapper instead of one of its own interfaces because the object uses its interfaces to access instance-specific data. An object could receive an interface wrapper to one of its own interfaces if a client passes an interface pointer back to the owning object as an input parameter on another call. The solution is simply to unwrap an inter-

face pointer whenever the pointer is passed as a parameter to its owning object.

4.5. Acquiring Static Interface Metadata

Interface wrapping requires static metadata about interfaces. The interface wrapper must be able to identify all interface pointers passed as parameters to an interface member function. There are a number of sources for acquiring static interface metadata. Possible sources include the MIDL description of an interface, COM type libraries, and interface proxies and stubs.

Acquiring static interface metadata from the MIDL description of an interface requires static analysis tools to parse and extract the appropriate metadata from the MIDL source code. In essence, it needs the MIDL compiler. Ideally, interface static metadata should be available to the interface wrapping code in a compact binary form.

Another alternative is to acquire static interface metadata from the COM type libraries. COM type libraries allow access to COM objects from interpreters for scripting languages, such as JavaScript [18] or Visual Basic [13]. While compact and readily accessible, type libraries describe only a subset of possible COM interfaces. Interfaces described in type libraries cannot have multiple output parameters. In addition, the metadata in type libraries does not contain sufficient information to determine the size of all possible dynamic array parameters.

Static interface metadata is also contained in the interface proxies and stubs. MIDL-generated proxies and stubs contain marshaling metadata encoded in strings of marshaling operators (called MOP strings). Static interface metadata can be acquired easily by interpreting the MOP strings. Unfortunately, the MOP strings are not publicly documented. Through an extensive process of trial and error involving more than 600 interfaces, at the University of Rochester, we were able to determine the meanings of all MOP codes emitted by the MIDL compiler.

Our interception system contains a MOP interpreter and a MOP precompiler. A heavyweight, more accurate interception subsystem uses our homegrown MOP interpreter. A lightweight interception subsystem uses the MOP precompiler to simplify the MOP strings (removing full marshaling information) before application execution.

The MOP precompiler uses dead-code elimination and constant folding to produce an optimized metadata representation. The simplified metadata describes all interface pointers passed as interface parameters, but does not contain information to calculate parameter sizes or fully walk pointer-rich arguments. Processed by a secondary interpreter, the simplified metadata allows the lightweight runtime to wrap interfaces in a fraction of the time required with full MOP strings.

While other COM instrumentation systems do use the MOP strings to acquire static interface metadata, ours is the first system to exploit a precompiler to optimize parameter access

The interception system acquires MOP strings directly from interface proxies and stubs. However, in some cases, components are distributed with MIDL source code, but without interface proxies and stubs. In those cases, the programmer can easily create interface proxies and stubs from the IDL sources with the MIDL compiler. OLE ships with about 250 interfaces without MOP strings. We were able to create interface proxies and stubs with the appropriate MOP string in under one hour using MIDL files from the OLE distribution.

4.6. Coping With Undocumented Interfaces

A final difficulty in interface wrapping is coping with *undocumented* interfaces, those interfaces without static metadata. While all documented COM interfaces should have static metadata, we have found cases where components from the same vendor will use an undocumented interface to communicate with each other.

When a function call on a documented interface is intercepted, the interface wrapper processes the incoming function parameters, creates a new stack frame, and calls the object interface. Upon return from the object's interface, the interface wrapper processes the outgoing function parameters and returns execution to the client. Information about the number of parameters passed to the member function is used to create the new stack frame for calling the object interface. For documented interfaces, the size of the new stack frame can easily be determined from the marshaling byte codes.

When intercepting an undocumented interface, the interface wrapper has no static information describing the size of stack frame used to call the member function. The interface wrapper cannot create a stack frame to call the object. It must reuse the existing stack frame. In addition, the interface wrapper must intercept execution return from the object in order to preserve the interface wrapping invariants used to identify objects and to determine interface ownership.

For function calls on undocumented interfaces, the interface wrapper replaces the return address in the stack frame with the address of a trampoline function. The original return address and a copy of the stack pointer are stored in thread-local temporary variables. The interface wrapper transfers execution to the object directly using a jump rather than a call instruction.

When the object finishes execution, it issues a return instruction. Rather than return control to the caller—as would have happened if the interface wrapper had not replaced the return address in the stack frame—

execution passes directly to the trampoline. As a fortuitous benefit of COM's callee-popped calling convention, the trampoline can calculate the function's stack frame size by comparing the current stack pointer with the copy stored before invoking the object code. The trampoline saves the frame size for future calls, and then returns control to the client directly through a jump instruction to the temporarily stored return address.

The return trampoline is used only for the first invocation of a specific member function. Subsequent calls to the same interface member function are forwarded directly through the interface wrapper.

By using the return trampoline, the interception system continues to function correctly even when confronted with undocumented interfaces. To our knowledge, our is the only COM instrumentation system to tolerate undocumented interfaces.

4.7. Evaluation of Interface Wrapping

Function vs. Interception Technique	IUnknown::AddRef	IStream::Read
Direct Call	0.19us	15.73us
Replace Interface Pointer	0.26us	16.24us
Replace VTBL	0.26us	16.24us
Replace Function Pointer	0.26us	16.24us
Intercept Object Code	0.30us	16.29us

Table 2. Interface Interception Times.

Listed are the times for intercepting the `IUnknown::AddRef` and `IStream::Read` (with 256 bytes of payload data) on a 200MHz Pentium PC.

Detailed in Table 2, wrapping the interface by replacing the interface pointer adds a 36% overhead to trivial function like `IUnknown::AddRef` and just a 3% overhead to a function like `IStream::Read`. Processing the function arguments with interpreted MOP strings adds on average about 20% additional execution overhead while processing with precompiled MOP strings adds under 3% additional overhead. Replacing the interface pointer is preferred over the alternative interception mechanisms because it does not break under common COM programming idioms.

5. Related Work

Brown [1, 2] describes an interception system for COM using Universal Delegators (UDs). To use Brown's UD, the application programmer is entirely responsible for wrapping COM interfaces. The programmer must manually wrap each outgoing or incoming parameter with a special call to the UD code. While providing robust support for applications such as object aggregation, Brown's UD is not suitable for binary-only interception and instrumentation.

HookOle [10] is a general interception system for instrumenting COM applications. Like our system, HookOle extracts interface metadata from MIDL MOP strings. However, rather than replacing interface pointers, HookOLE replaces function pointers (in the VTBL) and assumes that the same function will not be used to implement multiple, dissimilarly typed interfaces. HookOLE breaks whenever an object uses universal delegation. HookOle provides no support for undocumented interfaces. The ITest Spy Utility [14] uses HookOle to provide a test harness for OLE DB components.

Microsoft Transaction Server (MTS) [21] intercepts inter-component communication to enforce transaction boundaries and semantics. MTS wraps COM interfaces in a manner similar to our interception system. However, MTS supports only a subset of possible COM interfaces and does not provide support for undocumented interfaces.

COM+ [9] provides a generalized mechanism called, *interceptors*, for intercepting communication between COM+ objects. A significant redesign of COM, COM+ has complete control over the memory layout of all objects. This control significantly reduces the complexity of interception, but only works for newly designed COM+ components.

COMERA [24] is an extensible remoting architecture for distributed COM communication. COMERA relies on existing DCOM [3] proxies and stubs to intercept cross-process communication. Neither COMERA nor DCOM support in-process interception.

Eternal [17] intercepts CORBA IIOP-related messages via the Unix `/proc` mechanism. Intercepted messages are broadcast to objects replicated for fault tolerance. The `/proc` mechanism is limited to cross-process communication and extremely expensive (requiring at least two crossings of process boundaries).

Finally, a number of CORBA [23] vendors support interception and filtering mechanisms. In general, instrumenting COM applications is more difficult than equivalent CORBA applications. COM standardizes interface format, but not object format. Each ORB specifies parts of the CORBA object format related to interception. So for example, the interface ownership

problem has no equivalent in CORBA, but the problem of instrumenting binary CORBA application independent of ORB vendor remains unsolved.

6. Conclusions and Future Work

We have described a general-purpose interception system for instrumenting COM components and applications. Important features of our interception system include inline redirection of all COM object-instantiation functions, interception of COM interfaces through interface wrappers, accurate tracking of interface ownership, and robust support for undocumented interfaces.

Our interception system has been tested on over 300 COM binary components, 700 unique COM interfaces, and 2 million lines of code. Using our interception system, the Coign ADPS has automatically partitioned and distributed three major applications including Microsoft PhotoDraw 2000.

While our interception system is COM specific, the techniques described are relevant to CORBA ORBs. For example, inline redirection and interface wrappers could be used to intercept Portable Object Adapter (POA) [20] functions and object invocations [22].

Bibliography

[1] Brown, Keith. Building a Lightweight COM Interception Framework, Part I: The Guts of the UD. *Microsoft Systems Journal*, vol. 14, pp. 49, February 1999.

[2] Brown, Keith. Building a Lightweight COM Interception Framework, Part I: The Universal Delegator. *Microsoft Systems Journal*, vol. 14, pp. 17, January 1999.

[3] Brown, Nat and Charlie Kindel. *Distributed Component Object Model Protocol -- DCOM/1.0*. Microsoft Corporation, Redmond, WA, 1996.

[4] Hartman, D. Unclogging Distributed Computing. *IEEE Spectrum*, 29(5), pp. 36-39, May 1992.

[5] Hunt, Galen. Automatic Distributed Partitioning of Component-Based Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, 1998.

[6] Hunt, Galen. Detours: Binary Interception of Win32 Functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. *to appear*. Seattle, WA, July 1999.

[7] Hunt, Galen and Michael Scott. A Guided Tour of the Coign Automatic Distributed Partitioning System. *Proceedings of the Second International Enterprise Distributed Object Computing Workshop (EDOC '98)*, pp. 252-262. La Jolla, CA, November 1998. IEEE.

[8] Hunt, Galen C. and Michael L. Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pp. 187-200. New Orleans, LA, February 1999. USENIX.

[9] Kirtland, Mary. Object-Oriented Software Development Made Simple with COM+ Runtime Services. *Microsoft Systems Journal*, vol. 12, pp. 49-59, November 1997.

[10] Microsoft Corporation. *HookOLE Architecture.* Alpha Release, Redmond, WA, October 1996.

[11] Microsoft Corporation. *Internet Explorer.* Version 2.0, Redmond, WA, October 1997.

[12] Microsoft Corporation. Overview of the Corporate Benefits System. *Microsoft Developer Network*, January 1997.

[13] Microsoft Corporation. *Visual Basic Scripting Edition.* Version 3.1, Redmond, WA, October 1997.

[14] Microsoft Corporation. *ITest Spy Utility.* OLE DB SDK, Version 1.5. Microsoft Corporation,, Redmond, WA, January 1998.

[15] Microsoft Corporation. *PhotoDraw 2000.* Version 1.0, Redmond, WA, November 1998.

[16] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, Redmond, WA, 1995.

[17] Narasimhan, P., L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. *Proceedings of the Thrid USENIX Conference on Object-Oriented Technologies*. Portland, OR, June 1997.

[18] Netscape Communications Corporation. *Netscape JavaScript Guide*, Mountain View, CA, 1997.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* vol. Revision 2.0, Framingham, MA, 1995.

[20] Object Management Group. Specification of the Portable Object Adapter (POA). OMG Document orbos/97-05-15 ed. June 1997.

[21] Reed, Dave, Tracey Trewin, and Mai-lan Tomsen. Microsoft Transaction Server Helps You Write Scalable, Distributed Internet Apps. *Microsoft Systems Journal*, vol. 12, pp. 51-60, August 1997.

[22] Schmidt, Douglas C. and Steve Vinoski. Object Adapters: Concepts and Terminology. *C++ Report*, 9(11), November 1997.

[23] Vinoski, Steve. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), February 1997.

[24] Wang, Yi-Min and Woei-Jyh Lee. COMERA: COM Extensible Remoting Architecture. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pp. 79-88. Santa Fe, NM, April 1998. USENIX.

[25] Williams, Sara and Charlie Kindel. The Component Object Model: A Technical Overview. *Dr. Dobb's Journal*, December 1994.

Implementing Causal Logging using OrbixWeb Interception

Chanathip Namprempre Jeremy Sussman
Keith Marzullo
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA, 92093-0114
{cnamprem,jsussman,marzullo}@cs.ucsd.edu

Abstract

Some form of replicated data management is a basic service of nearly all distributed systems. Replicated data management maintains the consistency of replicated data. In wide-area distributed systems, *causal consistency* is often used, because it is strong enough to allow one to easily solve many problems while still keeping the cost low even with the large variance in latency that one finds in a wide-area network. *Causal logging* is a useful technique for implementing causal consistency because it greatly reduces the latency in reading causally consistent data by piggybacking updates on existing network traffic.

We have implemented a CORBA service, called COPE, that is implemented by using causal logging. COPE also shares features with some CORBA security services and is naturally implemented using the OrbixWeb interception facilities. In implementing COPE in OrbixWeb, we encountered several problems. We discuss COPE, its implementation in OrbixWeb, and the problems we encountered in this paper. We hope that this discussion will be of interest to both those who are implementing and who are planning on using CORBA interception facilities.

1 Introduction

In nearly all distributed systems, some data values are replicated across different processors. For example, one might have a distributed cache to allow reads to be performed more quickly. Or, one might have multiple copies of a critical data structure to ensure that should a processor crash or become isolated by a network failure, a copy of the data will still remain accessible. Replicated data management is therefore an essential service of distributed systems.

One issue arising in replicated data management is how *consistent* the replicas need to be. Considerable effort has gone into defining and analyzing different consistency models. A very strong requirement is for the replicas to reflect the real-time order in which they were written. For example, consider two values x and y that are replicated on processors a, b, c and d, and let the initial values of x and y be zero. If processor a sets x to 1 before processor b sets y to 2, then c and d will both see x set to 1 before seeing y set to 2 (the same, of course, holds for a and b). This kind of consistency, called *atomic consistency* [7], is in general expensive to implement, and thus used only when absolutely required. A weaker form of consistency is called *sequential consistency* [6] in which *some* total order on updates is imposed. With the above example, c and d will both see the same order of updates to x and y, but not necessarily the update of x before the update of y. Sequential consistency is less expensive to implement than atomic consistency and yet is often strong enough for many applications.

A still weaker consistency property is *causal consistency* [1]. Suppose that processor b did not update y until it read x and found a value of 1. In this case, we say that the value of y *causally depends* upon the value of x—that is, the act of a setting x to 1 in some sense caused b to set y to 2. Causal consistency ensures that the sequence of updates as read by other processors is consistent with causal dependency. In this case, both c and d would see the update of x before the update of y. On the other hand, if b did not read x before setting y, then the

updates are said to be *concurrent*. Concurrent updates are not ordered, and so c could see x updated before y while d could see y updated before x.

Preserving only causal dependencies among replicated data is sufficient for many applications [1]. We give an example of such an application later in this paper, namely optimistic execution in an object-oriented environment. And, causal consistency is cheaper to implement than sequential consistency in a wide-area setting. A problem with wide-area networks is the large variance in communication latency. With sequential consistency, a processor that updates a shared variable must ensure that its update is ordered with any other potentially concurrent updates, and so the latency of an update can be no smaller than the longest latency from the updating processor to a copy of the data [11]. Causal consistency does not require concurrent updates to be ordered and so a processor can simply update its local copy and continue. There can, however, be latency introduced when reading a shared variable [10].

Latency can be reduced by implementing causal consistency through a technique called *causal logging* [3]. With causal logging, a message that updates a shared variable piggybacks the updates upon which the variable causally depends. That is, causal logging trades off bandwidth for latency. Causal logging has been used in several applications besides implementing causally consistent replicated data, including distributed simulation [5] and techniques for low-cost failure recovery [2]. These applications all share the same general property: a process does not observe an action (such as the delivery of a message or the update of a shared variable) until it has observed all actions that the observed action causally depends upon.

In one of our research projects, we faced the problem of implementing causal logging when constructing a CORBA service. We call this service *COPE* and briefly describe it in Section 2. To implement this service in CORBA, we hoped to use an *interception* facility. Interception is a way to add functionality to CORBA services in a manner that is orthogonal and non-intrusive to the main computation. CORBA interception is implemented using *interceptors* which are code that can be invoked upon a message being sent, or upon a message being received (as well as other *trigger* points). We describe why this facility allows for a straightforward implementation of causal logging, as well as describing this implementation, in Section 3. We chose OrbixWeb as the platform for implementing COPE because it is a popular Java ORB that provides interception facilities. We detail these features in Section 4.

COPE is a somewhat complex CORBA service, and to the best of our knowledge no other group has considered a service with similar functionality. It shares some features with proposed CORBA security services, most notably *Application Access Policy* [9]. In implementing COPE, we encountered several problems with OrbixWeb, which we describe in Section 5. We believe that the problems we encountered will also be encountered by those implementing similar CORBA services. Our goal in writing this paper is to discuss the problems we encountered in hope that those building CORBA ORBs will be aware of them when building interception facilities.

2 COPE

We have implemented a new CORBA service called COPE that is based on causal logging. We give a brief overview of COPE to ground the discussion in Section 3 on what we require of a CORBA causal logging implementation.

One of the two abstractions that COPE implements is the class of *assumptions*. An assumption is a CORBA object that is eventually either asserted or refuted. An assumption keeps track of the objects that wish to be notified when it is resolved. Assumptions can also be subclassed to associate semantics with them, such as assumptions that depend on other assumptions. An example of such an assumption is a *proposition* which is expressed as a boolean formula over a symbol table. Each entry in the symbol table is itself an assumption. A proposition assumption becomes asserted or refuted when the value of its formula evaluates to *true* or *false* as determined by the assumptions that have been asserted and refuted in the symbol table.

The other abstraction that COPE implements is the class of *optimists*. An optimist is a CORBA object that takes on assumptions. Optimists can execute *optimistically* based on the assumptions that it has taken on. More specifically, an optimist can either checkpoint its state when it takes on an assumption or it can block, awaiting the eventual assertation or refutation of the assumption. If, in either case,

the assumption is asserted then the optimist is notified so that it can either discard the checkpoint or continue execution. If, on the other hand, the assumption is refuted, then the optimist is notified so that it can either roll it state back to the associated checkpoint or continue execution knowing that the assumption was refuted.

Assumptions are causally consistent with respect to CORBA communications. For example, consider optimist a invoking method $b.m$ on optimist b. If a has taken on an assumption x which is still unresolved by the time a invokes $b.m$, then b must take on x by the time it begins execution of $b.m$. Similarly, if $b.m$ constructs an optimist c, then c must also take on x by the time c completes initialization.

Put into terms of shared memory, each optimist has a list of replicas of unresolved assumption. These replicas are causally consistent, where "causally depends" is defined in terms both of optimists invoking methods on other optimists and of optimists creating new optimists. This list is maintained as follows:

1. When an optimist a makes a method invocation on an optimist b, it piggybacks on the method invocation a list of unresolved assumptions that a has taken on.

2. When an optimist b has a method invoked by an object a, b checks to see if a has a class that derives from *Optimist*. If so, then b strips off any assumptions piggybacked on the method invocation and decides whether to add them to its own list of unresolved assumptions or to block the invocation.

3. When an optimist a creates an optimist b, it makes a method call to an *optimist factory*. As when invoking a method on an optimist, a piggybacks on the method invocation a list of assumptions that a has taken on. The factory f checks to see if a has a class that derives from *Optimist*. If so, it then passes a's unresolved assumptions to b. Optimist b can choose either to accept a's assumptions, in which case the creation is successful, or to deny them, in which case b is not created.

These three rules for maintaining the list of assumptions together implement causal logging of assumptions. Other features of COPE, such as assertion resolution and object notification, are not germane to the discussion. Interested readers can find further details on COPE in [8].

3 Implementing Causal Logging Using Interception

Consider implementing causal logging on top of CORBA. The following properties of an implementation state what we believe constitutes a well-engineered solution.

- **Transparency**. The piggybacking and stripping of piggybacked information should be implemented without explicit involvement of the objects using causal logging. To do otherwise would make it hard to ensure that the causal logging mechanism is correct.

- **Scheduling**. Causal logging implies that information is made available to an object at certain points in its execution. To do otherwise might violate the causal consistency condition. Hence, the causal logging mechanism notifies of the delivery of causal information ordered with respect to the invocation of methods and creation of new objects.

- **Context Sensitivity**. The information that an object a piggybacks to an object b may depend both on the state and class of a and on the class of b. Without knowledge about a, it is hard to piggyback *any information* because there is no way to obtain it, and without information about b an object would have to piggyback all possibly useful information on every method invocation. The latter would both be inefficient and would pose a possible security problem.

CORBA interception is ideally suited as a piggybacking mechanism that provides the properties of transparency and scheduling. Interceptors are orthogonal to the regular path of computation, and therefore provide transparency. Furthermore, since interception can be placed at many points in the method invocation sequencing, it can be used to provide scheduling as well.

A simple implementation of causal logging would be as follows. Consider an interception mechanism in

which every object has an interceptor that is specific to that object. The interceptor knows the identity of the object with which it is associated, and the interceptor is invoked upon both ends of a method invocation—that is, by the invoked object and by the invoking object.

When a method is invoked, the interceptor on the invoking object uses some mechanism to determine what type of information is to be piggybacked. This mechanism can base its determination on the class of the invoked object. The actual information can be determined from the current state and the class of the invoking object. Hence, context sensitivity is implemented. The interceptor adds this data to the outgoing method invocation.

When the invoked object receives the invocation, its interceptor removes the data that was added by the invoking object's interceptor. The invoked object's interceptor then implements scheduling by ordering the method invocations that deliver the causally logged information with respect to the incoming method invocation.

This scheme implements encapsulation of the method invocations within the causal logging mechanism. That is, the underlying method invocations are not altered, but rather are used as a conduit of causally logged information and are scheduled to maintain causal consistency. The CORBA interception facility is intended for exactly this kind of encapsulation. Our simple model of CORBA interception requires the following capabilities:

1. Interceptors should be invoked on all outgoing and incoming invocations.

2. Interceptors should be able to add information when a method invocation is initiated at the invoking side and remove information when the method invocation is initiated at the invoked side.

3. An interceptor on the invoking side should know the state and class of the object with which it is associated and the class of the object being invoked.

4. An interceptor on the invoked side should have the ability to make method calls on the object with which it is associated before it allows the initiating method to be invoked.

As discussed in Section 5, we had a few difficulties in creating such an architecture in OrbixWeb.

4 OrbixWeb

In COPE we implement interception by using OrbixWeb filters. OrbixWeb is an implementation of CORBA by IONA Technologies Inc. It is compliant with the Object Management Group's (OMG) CORBA specification Version 2.0 and is implemented in Java. It provides a filtering mechanism with support for piggybacking of additional information onto method invocations. There are two types of filters in OrbixWeb: *per-process* and *per-object* filters. We describe both types in turn.

4.1 Per-Process Filters

A per-process filter is code that is associated with a client or server process. The filter monitors all incoming and outgoing method invocations and attribute references that reference objects associated with another process. More than one process filter can be chained together to the same process. There are ten points where code in a filter can be associated with a process: inRequestPreMarshal, outRequestPreMarshal, inReplyPreMarshal, outReplyPreMarshal, inRequestPostMarshal, outRequestPostMarshal, inReplyPostMarshal, outReplyPostMarshal, inReplyFailure, and outReplyFailure. Figure 1 (taken from [4]) illustrates these filter points.

The name of a filter method indicates where in the method invocation sequence it is invoked. The modifier *request* or *reply* indicates whether the filter is associated with the invocation of the method or the reply from the method. The modifier *in* or *out* indicates the direction the method invocation or method reply is going with respect to the process with which the filter is associated. In particular, *out* indicates the invoking object for method invocations and the invoked object for method reply, and *in* indicates the invoked object for method invocations and the invoking object for method reply. The stem indicates exactly where in the processing of the method invocation the filter is associated: *PreMarshal* is before parameter marshalling, *PostMarshal* is after parameter marshalling. *Failure* is for exceptions. Specifi-

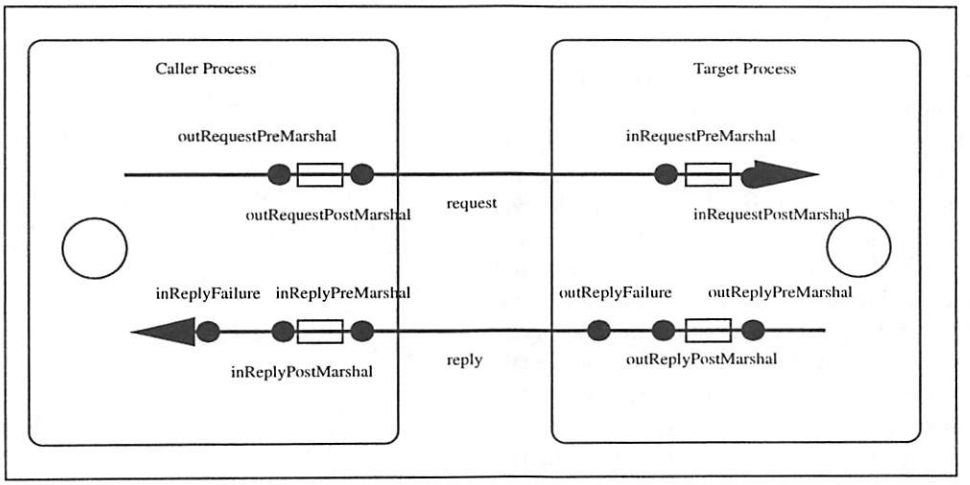


Figure 1: Per-process filter monitor points

cally, code that is associated with failure filter points is executed under two conditions: (1) when an exception condition is raised by the target of the invocation or (2) when there are return values from any preceding filter points indicating that the call is not to be processed any further.

The OrbixWeb abstract class `IE.Iona.OrbixWeb.Features.Filter` implements per-process filters. A user-defined filter is implemented by defining a class that inherits from the `Filter` class. When a process creates an object of this class, the newly-created filter is associated with the creating object's process. Successive filter creation results in these filters being chained in the order of their creation.

The following demonstrates the construction of a per-process filter [4]:

```
1  public class ProcessFilter extends Filter {
2  public boolean outReplyPreMarshal(Request r)
   {
3   String s, o;
4   long l = 27;

5   try {
6    s = ORB.init().object_to_string(
         (r.target()));
7    o = r.operation ();

8    OutputStream outs =
9    _OrbixWeb.Request(r).create_output_stream();
10   outs.write_long (l);
11  } catch (SystemException se) {
12    System.out.println("Caught exception "+se);
    }

13  System.out.println ("Request to "+ s);
14  System.out.println ("with operation "+ o);
15  return true; // continue the call
    }
   }
```

Information about the invocation such as the target, operation name, and arguments can be accessed through the parameter `r`, which is of type `IE.Iona.OrbixWeb.CORBA.Request`. For example, the call `r.target()` in Line 6 of the code above returns the target of the invocation while the call `r.operation()` in Line 7 returns the name of the operation being invoked.

As mentioned previously, OrbixWeb also allows extra information to be piggybacked onto the method invocation. The call `_OrbixWeb.Request(r).create_output_stream()` in Line 9 above creates a stream to which extra information can be written (`outs.write_long(l)` in Line 12). This information can later be read by the corresponding filter point on the other side of the invocation (in this example, an `inReply` filter on the client side).

4.2 Per-Object Filter

Filters can be associated with a given object as well. To define a per-object filter, one defines a Java object that implements the Java interface for the CORBA object that was generated by the IDL compiler. This new Java object implements the desired filters. For example, assume the CORBA object a provides a method m. The IDL compiler generated Java interface includes the declaration of method m. Any per-object filter that can be associated with the class a must implement this interface, including the method m. The association is done by having the implementation of a create an instance of the per-object filter object, and then specify where in the method invocation path the filter is to be invoked. The following demonstrates the association of two per-object filters `filter1` and `filter2` with an object of class `Foo`:

```
1   Foo foo;
2   (( _FooSkeleton) foo)._preObject
       = filter1;
3   (( _FooSkeleton) foo)._postObject
       = filter2;
```

This mechanism for implementing a per-object filter is simple and elegant. However, as will be discussed in Section 5.2, it is too restrictive for our purposes.

5 Problems

In building COPE, we encountered difficulties in using OrbixWeb. This section describes two of these difficulties:

1. Representing the CORBA inheritance of an object in the underlying Java implementation.

2. The lack of support for a generic per-object filter.

We also give our corresponding workarounds and evaluate their effectiveness.

5.1 CORBA Inheritance Issues

This problem arose because COPE piggybacks references to assumption objects, and it is common to derive specific kinds of assumptions from the assumption class. It is always somewhat complex figuring out how to implement a CORBA-based program using a specific ORB, but figuring out how to structure COPE was especially hard. It took approximately a month of mail exchange with Iona before the problem was understood well enough to resolve.

Most ORBs are implemented using an object-oriented language, such as Java or C++. A CORBA-based application is defined, in IDL, as a set of classes that may be related via single inheritance. The IDL compiler translates these inheritance relations in some manner into a set of class definitions in the implementation language. Consider a CORBA class `Bar` that inherits from a CORBA class `Foo`. The implementation object (say, `BarImpl`) should also inherit from the implementation object `FooImpl`. In addition, with most IDL compilers both implementation objects are instances of a base CORBA object class.

The problem discussed here is concerned with the translation chosen by the OrbixWeb IDL compiler.[1] Before discussing the problem further, we provide some background concerning how objects are implemented in OrbixWeb. Suppose we have an IDL interface as follows:

```
interface Foo {
   void foo_method();
};

interface Bar : Foo {
   void bar_method(Foo f);
};
```

The OrbixWeb IDL compiler generates eight files for each CORBA interface. For example, the CORBA interface `Foo` is compiled into `FooHolder`, `FooHelper`, `_FooSkeleton`, `_FooStub`, `_FooImplBase`, `_tie_Foo`, `_FooOperations`, and `Foo`. The first two are helper classes that support marshalling, narrowing, and other CORBA support

[1] Recall that Java supports only single inheritance, and so a translation based on multiple inheritance is only possible through interfaces, not classes.

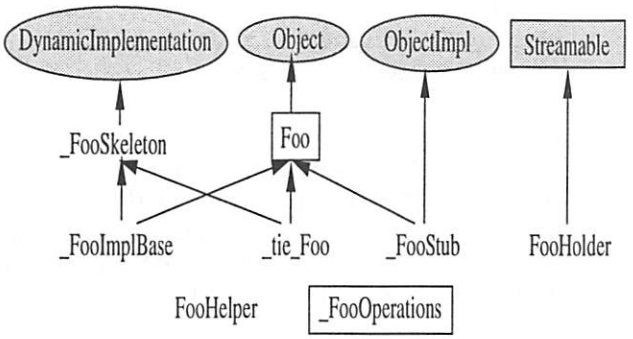


Figure 2: IDL-generated files for `Foo`

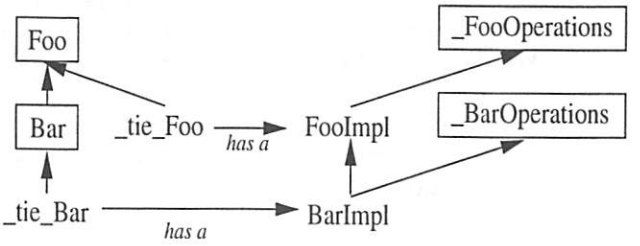


Figure 3: Inheritance Diagram

operations. The next two are classes that are placeholders for the stubs. The last four, two classes and two interfaces respectively, are described in more detail below.

The complete inheritance hierarchy for `Foo` is depicted in Figure 2. Rectangles represent interfaces. Shaded ovals and rectangles represent classes and interfaces provided in standard packages such as `org.omg.CORBA`. These packages are part of OrbixWeb core classes.

There are two approaches with which one can implement an OrbixWeb object: the "ImplBase" approach and the "tie" approach [4]. Suppose you wish to implement the CORBA class `Foo` with the Java class `FooImpl`. In the ImplBase approach, `FooImpl` has the following signature:

```
public class FooImpl extends _FooImplBase
```

That is, `FooImpl` is a subclass of the class `_FooImplBase`. And, since `_FooImplBase` implements the Java interface `Foo` (See Figure 2), `FooImpl` also implements `Foo`. Therefore, a CORBA `Foo` object can be instantiated as follows:

```
Foo foo = new FooImpl();
```

The tie approach, in contrast, is a delegation model. With this approach the `FooImpl` class implements the `_FooOperations` interface. However, since `_FooOperations` and `Foo` are both interfaces, to instantiate a CORBA `Foo` object one first instantiates a `FooImpl` instance and then instantiates a `_tie_Foo` instance with the `FooImpl` instance as a parameter:

```
Foo foo = new _tie_Foo( new FooImpl() );
```

Since `_tie_Foo` implements `Foo`, the variable `foo` has type `Foo` as desired.

Now consider implementing the class `Bar`. One would naturally wish the implementation `BarImpl` to inherit from `FooImpl`. But, `BarImpl` also implements the methods declared in the CORBA `Bar` class. As was done in implementing `Foo`, we can use either the ImplBase approach or the tie approach. However, the ImplBase approach requires `BarImpl` inheriting from `_BarImplBase`. This implies multiple inheritance of classes, which Java does not support. Thus, one is constrained to use the tie approach, viz.:

```
public class BarImpl extends FooImpl
   implements _BarOperations
```

Since we are using the tie approach, an instance of `Bar` is created by wrapping an instance of `BarImpl` in an instance of `_tie_Bar`:

```
Bar bar = new _tie_Bar( new BarImpl() );
```

Figure 3 illustrates the resulting inheritance diagram of a `FooImpl` object and a `BarImpl` object.

The problem with the tie approach, however, is that the implementation objects (`FooImpl` and `BarImpl` in this example), are not instances of the Java interface that represents the CORBA object (`Foo` and `Bar` in this example). This poses a problem when using CORBA operations.

For example, suppose that there is a CORBA `Bar` object `B1` on processor 1 and a CORBA `Bar` object `B2` on processor 2. `B2` has a reference `r` to `B1`, and invokes the method `r.bar_method(this)`.

Further suppose that the implementation of `bar_method` in `BarImpl` tests the parameter `f` to see if it is an instance of `Bar`:

```
1  public bar_method (Foo f) {
2    if (f instanceof Bar)
3       ...
4    else
5       ...
     }
```

One might expect that the code in Line 3 would be executed, but it is not due to an implementation decision by Iona. While marshalling `this` on `B2`, CORBA determines the class of the value it is marshalling through a method on it named `type`, e.g., it invokes `this.type()`. If `this` were to implement the Java interface `Bar`, then `this.type()` would return a value indicating the CORBA class `Bar`. But, since `this` implements `_BarOperations`, `this.type()` returns the value `null`. The marshalling code therefore declares the parameter passed in the message to `B1` to be of type reference to `Foo`.

We dealt with this problem in the manner recommended by Iona. We save in the Java implementation of every CORBA object a reference to the tie object. For example, let the member variable referring to the tie object of an instance of `Bar` be `tieObject`. The declaration of `tieObject` and the constructor in the definition of the class `BarImpl` can be as follows:[2]

```
public class BarImpl
  implements _BarOperations {
  protected Bar tieObject; // declaration

  BarImpl() {
     ...
     tieObject = new _tie_Bar( this );
     }
  }
```

[2]Note that the setting of `tieObject` must be the last member variable initialization. If not, then the member variables of the tie object may be initialized to incorrect values.

Then, `B2` invokes `r.bar_method(tieObject)` instead of `r.bar_method(this)`. Since `tieObjects` implements `Bar`, `tieObject.type()` returns a value indicating the CORBA class `Bar`.

This problem occurs in other situations. It is in general a good OrbixWeb design practice to have objects like `BarImpl` implement a method that returns a reference to the tie object. This reference should be used in all places where a reference to the implementation is passed using CORBA.

This additional complexity in structure can be avoided by enforcing a file naming structure on the user's code. For example, some IDL compilers generate a file that the user edits to include the Java implementation of the CORBA object. Unlike OrbixWeb, the IDL compiler knows the name of the implementation class when it generates the files. Hence, the IDL compiler can generate files that explictly inherit from this class as needed. In our example, if the IDL compiler names the implementation class for `Foo` as `FooObj`, then the implementation class for `Bar` might be generated as

```
public class BarObj
   extends FooObj
   implements _BarOperations
```

5.2 Per-Process Filters

The filters that implement COPE's causal logging perform the same operations for each method of an optimist object: they add assumptions to a method invocation on the invoking object's side and remove the assumptions on the invoked object's side. Ideally, one would like to be able to associate the same filter with each method of any class that derives from optimist, and this association should be done in a general way. Unfortunately, this cannot be done in OrbixWeb. As discussed in Section 4.2, an object that implements per-object filters is required to implement all methods defined in the IDL definition for the class a of objects with which it is associated. Classes that inherit from a require their own filters to be explicitly implemented.

Since a general purpose filter cannot be constructed as a per-object filter and since we are unwilling to change the OrbixWeb IDL compiler, a per-process filter is our only option. A per-process filter is invoked for all method invocations leaving and enter-

ing the process, and so it can be used to implement a generic filter. However, using per-process filters raises other problems. We have found workarounds for these problems, but we do not believe that the workarounds are acceptable in terms of meeting our engineering requirements.

5.2.1 Performance

There are performance reasons leading one to implement several objects in the same process. It is relatively inexpensive for these objects to invoke each other's methods, since such an invocation does not require a context switch. Furthermore, there are serious problems in resource utilization with running multiple Java virtual machines on the same processor. Hence, one often tries to structure an OrbixWeb application with as many objects as possible in the same process.

However, per-process filters are not invoked for method invocations between objects in the same process. Hence, per-process filters can not be used to implement causal logging among objects in the same process. This implies that each COPE optimist must run in its own process. This solution carries an enormous performance penalty.

5.2.2 Lack of Required Information

An OrbixWeb filter can obtain various information about the invocation that it is intercepting. This information includes the reference of the object whose method is being invoked, the name of the method being invoked, the parameters of the method being invoked, and the name of the user running the program that resulted in the invocation. Unfortunately, the filter cannot determine the reference of the object making the invocation.

Without this information, it is impossible to provide the context sensitivity property defined in Section 3. The reason is that the piggybacked data depends on the state of the invoking object. Thus, the filter cannot obtain the information to be piggybacked from the invoking object.

A per-object filter *is* aware of the object with which it is associated, and so does not suffer from this problem. However, as we saw in Section 5.2, per-object filters cannot be used. Hence, we needed to find a workaround.

Because of the constraint of having only one object per process, we work around this problem by allocating a static variable that contains a reference to the tie object. This static variable is referenced by the filters when they need to make a invocation on the invoking process. This is a simple workaround, but it is artificially simple because of the constraint of one object per process. If the per-process filter could be imposed for communications between objects in the same process, then this static variable would need to be updated before every method invocation. Doing so would violate transparency of Section 3.

6 Discussion

In implementing causal logging to achieve causal consistency for our CORBA service, COPE, we use the interception facilities provided by OrbixWeb filters. OrbixWeb filters allow us to add functionality to legacy software in a manner that is orthogonal and non-intrusive to the main computation. Using filters, invocations in the system can be captured and processed before continuing with the normal flow of the program. However, despite our best endeavors, we were faced with a few difficulties. One problem was in understanding how to structure an OrbixWeb application that passes references to CORBA objects that can be subclassed. With the help of Iona, we were able to find a practical solution. The two remaining problems were more serious:

1. One cannot impose a per-object filter that is generic—that is, that need not conform to the interface implemented by the object. The level of interception implementable using OrbixWeb filters therefore poses a fundamental problem. As we discovered in our case, getting around the problem incurs a very high performance penalty and is thus not a practical solution.

2. One has no means to access the calling object in a filter. This violation of context sensitivity property leads to the static variable solution which, except for the problem above, would violate the transparency property.

It is not clear why such a limit to disallow access to the calling object was imposed in the first place. But with this limitation, the piggybacking mechanism in place cannot be utilized to its full potential.

We believe that any communications middleware platform for distributed computing should be powerful enough to build the engineering solution described in Section 3. Such a solution might even be considered a benchmark for the utility of such platforms.

Although we have not attempted to implement COPE using other CORBA ORBs, we have looked into using the Legion system [12]. And, Legion appears to be powerful enough to efficiently implement COPE, but it would be interesting to actually do so to see what problems might arise. Of course, it would also be interesting to see how well other CORBA ORBs could support COPE.

7 Acknowledgments

We would like to thank Treasa O'Shaughnessy, Anne O'Shea and John O'Shea of IONA who helped us work through the technical problems we have encountered with OrbixWeb. We were unable to resolve the problem described in Section 5.1 without their help. We benefited from discussions with Ang Nguyen-Toung at the University of Virginia of how COPE might be implemented on Legion. The COPE architecture was developed by the authors and Ramanathan Krishnamurthy and Aleta Ricciardi, both of the University of Texas at Austin.

This work was supported by the Defense Advanced Research Projects Agency (DoD) under contract number F30602-96-1-0313. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

8 Availability

The source of COPE is available from the COPE home page at the following URL:

`http://www.cs.ucsd.edu/users/marzullo/COPE.html`

References

[1] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. In *Distributed Algorithms, Fifth International Workshop WDAG '91*, pages 9–30, October 1991.

[2] Lorenzo Alvisi and Keith Marzullo. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[3] K. P. Birman and T. Joseph. Reliable communications in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] IONA Technologies Inc., Dublin, Ireland. *OrbixWeb Programmer's Guide*, 1997.

[5] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[6] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Keith Marzullo, Chanathip Namprempre, Jeremy Sussman, Ramanathan Krishnamurthy, and Aleta Ricciardi. Combining optimism and intrusion detection. Technical Report TR CS98-605, University of California, San Diego, Department of Computer Science and Engineering, October 1998.

[9] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: Common Object Request Broker Architecture*. Prentice Hall Press, October 1995.

[10] Michel Raynal, Andre Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[11] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[12] C. Viles, M. Lewis, A. Ferrari, A. Nguyen-Toung, and A. S. Grimshaw. Enabling flexibility in the Legion run-time library. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)*, pages 265–274, June 1997.

Quality of Service Aware Distributed Object Systems

Svend Frølund
Hewlett-Packard Laboratories, frolund@hpl.hp.com

Jari Koistinen
Commerce One, Inc. jari.koistinen@commerceone.com

Keywords: QoS-specification, QoS-enabled Trading, Distributed Object Systems, Object Component Specification, Object Interoperability

Computing systems deliver their functionality at a certain level of performance, reliability, and security. We refer to such non-functional aspects as quality-of-service (QoS) aspects. Delivering a satisfactory level of QoS is very challenging for systems that operate in open, resource varying environments such as the Internet or corporate intranets. A system that operates in an open environment may rely on services that are deployed under the control of a different organization, and it cannot per se make assumptions about the QoS delivered by such services. Furthermore, since resources vary, a system cannot be built to operate with a fixed level of available resources. To deliver satisfactory QoS in the context of external services and varying resources, a system must be QoS aware so that it can communicate its QoS expectations to those external services, monitor actual QoS based on currently available resources, and adapt to changes in available resources.

A QoS-aware system knows which level of QoS it needs from other services and which level of QoS it can provide. To build QoS-aware systems, we need a way to express QoS requirements and properties, and we need a way to communicate such expressions. In a realistic system, such expressions can become rather complex. For example, they typically contain constraints over user-defined domains where constraint satisfaction is determined relative to a user-defined ordering on the domain elements. To cope with this complexity we are developing a specification language and accompanying runtime representation for QoS expressions. This paper introduces our language but focuses on the runtime representation of QoS expressions. We show how to dynamically create new expressions at runtime and how to use comparison of expressions as a foundation for building higher-level QoS components such as QoS-based traders.

1. Introduction

Enterprises increasingly rely on distributed computer systems for business-critical functions. They often use such systems for internal information sharing, handling of business tasks such as orders and invoices, and accounting. In addition, businesses increasingly rely on distributed systems for their interactions with other business such as partners, customers, and sub contractors.

Since distributed systems are business critical, they must not only provide the right functionality, they must also provide the right quality-of-service (QoS) characteristics. By QoS, we refer to non-functional properties such as performance, reliability, quality of data, timing, and security. For some applications, best-effort QoS is acceptable; while others require predictable or guaranteed levels of QoS to function properly. In real-time systems, for example, timing is essential for correctness. In banking systems, security is necessary and must not be compromised. Business-critical enterprise systems and telecommunications systems must be highly available.

Ideally, we would like all systems to be up 100% of the time, be fully secure, and deliver exceptional performance. Unfortunately, building such systems is not realistic. In practice, we need to make trade-offs between QoS and cost of development and between different QoS categories. For example, achieving very high reliability is not only technically difficult, it is also very costly. Furthermore, providing very high reliability will also impose a performance overhead. QoS requirements, such as reliability, cannot be considered in isolation, they must be considered in the context of development cost and other QoS requirements, such as performance and security.

It is also common that a technology for satisfying one QoS aspect will not be compatible with a technology for satisfying another QoS aspect. As an example, it might be difficult to combine a group communication mechanism for high availability with certain security mechanisms.

To find the right solution for an enterprise computing system, we need to understand the cost of not satisfying certain QoS requirements and the cost of implementing and using mechanisms that provide a specific QoS level. To complicate things further, the cost of unsatisfactory QoS characteristics may vary according to the time of day, day of the week, and week of the year. It is also the case that the relative importance of specific QoS characteristics may vary over time. During day time availability might be more important than performance due to online sales transaction processing. During the night, accounting functions might need maximum performance to finish within a certain time.

In summary, we believe that enterprises will become increasingly dependent on distributed systems both for internal business automation and for the interaction with other enterprises and end customers. This will not only require the right functionality to be provided but also that the systems provide adequate quality-of-service. There are many issues in building systems with adequate QoS, and we believe that QoS must be considered systematically throughout the life-cycle of distributed enterprise systems. The goal is to support QoS-enabled systems, and we present a QoS fabric that is an essential building block for such systems.

1.1 QoS-Enabled Systems

A *QoS-enabled system* can provide defined levels of QoS to its users. Thus, users can customize the QoS delivered based on their preferences. Moreover, a QoS-enabled system can adapt to the environment in which it operates. For example, it can degrade gracefully if resources are scarce. There are many aspects to building QoS-enabled systems:

- **Mechanisms:** Different QoS levels are implemented by different mechanisms. Example mechanisms are reliability mechanisms, such as group communication protocols, and security mechanisms, such as specific encryption protocols. Different mechanisms must coexist in a QoS-enabled system, and it must be possible to select specific mechanisms based on the QoS level required from the system and the QoS level provided by the environment.
- **QoS Awareness:** A QoS-aware system is one that knows which levels of QoS it can deliver to its users and which levels of QoS it requires from its environment. Moreover, a QoS-aware system is able to communicate those QoS specifications to other entities.
- **QoS Agreements:** To deliver predictable QoS levels, it is necessary for systems to establish agreements with other systems about delivered QoS levels. Besides being able to describe and communicate QoS requirements and properties, QoS agreements also require trading and negotiation over these requirements and properties.
- **Monitoring:** The ability to monitor the QoS that is provided and received and to check compliance with existing agreements.
- **Meta Data:** Information about current load, number of deals, and available resources.

Today, most systems are not QoS enabled. Instead, they provide ad hoc or best-effort QoS. Sometimes special security, reliability, or other mechanisms are used, but applications are still unaware of the QoS that is required and provided. Figure 1 illustrates the dependencies of the different aspects of QoS enabling.

First one needs a variety of mechanisms—such as reliability and encryption protocols—that enable a system to satisfy QoS requirements. The installation of different mechanisms allows a system to provide different levels of QoS and adapt to the level of QoS provided by the environment. However, systems cannot install arbitrary mechanisms. Mechanisms can be changed within the constraints of the overall systems architecture. For example, if a system is built according to a particular security architecture, we may only be able to switch between a few reliable transports and encryption technologies that are compatible with that security architecture. Furthermore, adaptation is much simpler for clients than it is for servers. As an example, let a server and a client communicate over raw TCP/IP. If we want to increase the availability of the server, we might decide to switch to a group communication protocol. From the clients' perspective, this can be done quite transparently. For the server, however, it will have a significant impact. To enable group communication, servers must have functionality to join groups and transfer state that are not required in non-replicated systems. A system is QoS-enabled if it can adapt in an informed manner within the constraints of its overall system architecture.

We also need the ability to establish QoS agreements between the various systems and system entities, and we need the ability to monitor compliance of QoS agreements. Establishing and monitoring QoS agreements require that we can formalize these agreements. To that end, we need systems to be QoS aware. We also need meta data to adjust agreements based on current load and available resources. This paper focuses on a language and runtime representation to make distributed systems QoS aware.

1.2 QoS-Aware Systems

QoS-enabled systems must be QoS aware so that we can establish QoS agreements, both internally between the various system components, and externally between a system and its environment.

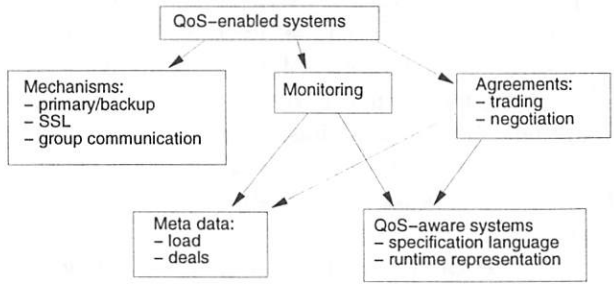


FIG. 1. Dependencies for QoS Enabling

For example, consider a distributed currency trading system. The front-end component of the system presents a user interface to human currency traders. The front-end uses a rate service to get rate updates and a currency trading service to perform currency trades. In order for the front-end to deliver predictable QoS to its human users, it must receive predictable QoS from the rate service and currency trading service. For example, the front-end may expect the rate service to be up 99 % of the time, deliver rate updates every minute, and provide information for a specific set of currencies. If the front-end is designed to work with a particular rate service, these expectations can be incorporated into the overall system design. However, if the front-end connects to a rate service dynamicall, it needs to be explicit about these expectations and establish a QoS agreement with a rate service based on these expectations.

Figure 2 illustrates the structure of this simple system and how QoS information need to flow dynamically in the system.

To facilitate the establishment of QoS agreements in distributed object systems, we need a way to specify the QoS requirements of clients (such as the front-end of the currency trading system) and the QoS properties of services (such as the rate service). We also need a way to communicate these specifications and compare them to determine if a particular service meets the requirements of a particular client.

To communicate QoS specifications between distributed objects, specifications must be represented as data structures at runtime. It is possible for each programmer to invent his own data format and construct QoS specifications in an ad-hoc manner by manually building up the appropriate data structures. Although possible, this is tedious, prone to error, and does not support interoperability.

Constructing QoS specifications in an ad-hoc manner is tedious and complicated because of the expressive power required and because we need to compare specifications to determine if one satisfies another. In terms of expressive power, QoS specifications essentially consist of constraints. However, the structure of these constraints is fairly complex. We need constraints over user-defined domains with a user-defined ordering, and we need constraints over statistical properties, such as mean, variance, and percentiles. Moreover, we need to bind these constraints to fine-grained entities, such as operation arguments, and to coarse-grained entities, such as interfaces. The required expressive power also makes it hard to compare specifications in an ad-hoc manner. For example, the comparison algorithm must operate on user-defined domains and take user-defined orderings into account.

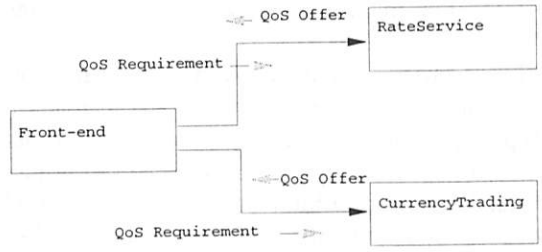


FIG. 2. Structure of the currency trading system

The construction of QoS-enabled, open systems will require a common specification technique and format analogously to the IDL and IIOP standards. Rather than have each programmer invent his own format, it is important to develop a common representation that allows interoperation.

We have defined a runtime format for QoS specifications and a runtime library to construct and manage such specifications. We refer to this format and library as a QoS fabric, and we call our QoS fabric QRR (QoS runtime representation). The runtime format is defined in terms of CORBA IDL types, and the runtime library is programmed in C++. However, QRR is not inherently tied to C++ or CORBA IDL, we could also implement the QRR fabric in JAVA and other languages, and DCOM and other distributed object infrastructures. We could even represent QML instances as XML documents.

Instantiating QRR specifications by hand as IDL-defined data structures is tedious. To allow QoS specifications to be written at a higher level, we have defined QML (QoS modeling language) and a QML to QRR compiler. To make it practical, we have integrated QML with existing technologies for distributed systems, such as interface definition languages, and design languages (UML).

The paper is organized as follows. Section 2 introduces QML and its underlying concepts for QoS specification. We then describe QRR in Section 3.1. We outline how we represent the QML concepts in terms of C++ and CORBA IDL, and we show the architecture of the QRR QoS fabric. We illustrate how to use QRR to implement distributed object systems with predictable QoS in Section 4. Finally, we give a brief overview of related work in Section 5, and we conclude in Section 6.

2. QML: A Language for QoS Specification

QML is a general-purpose QoS specification language; it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS category, such as reliability or performance. QML captures the fundamental concepts involved in the specification of QoS properties. Here, we give a brief introduction to these fundamental concepts. For a complete QML lan-

guage definition, including formal syntax and semantics, consult [5].

QML has three main abstraction mechanisms for QoS specification: *contract type*, *contract*, and *profile*. A contract type represents a specific QoS category, such as performance or reliability. Contract types a user-defined abstractions, there are no built-in contract types in QML. A contract type defines the *dimensions* that can be used to characterize a particular QoS category. A dimension has a domain of values that may be ordered. There are three kinds of domains: *set* domains, *enumerated* domains, and *numeric* domains. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QML profiles associate contracts with *interface entities*, such as operations, operation arguments, and operation results.

We use the currency trading example from the introduction to illustrate the QML specification mechanisms. We show how to specify QoS properties for a rate service object. Figure 3 gives a CORBA IDL [14] interface definition for a rate service object. It provides an operation, called `latest`, for retrieving the latest exchange rates with respect to two currencies. It also provides an operation, called `analysis`, that returns a forecast for a specified currency. The interface definition specifies the syntactic signature for a service but does not specify any semantics or non-functional aspects. Using QML, we can specify the QoS properties for this interface.

The QML definitions in Figure 4 include two contract types `Reliability` and `Performance`. The `Reliability` contract type defines three numeric dimensions. The first dimension (`numberOfFailures`) represents the number of failures per year. The keyword "decreasing" indicates that a smaller number of failures is better than a larger one. We use this dimension semantics to compare specifications under a "stronger than," or conformance, relation. Time-to-repair (`TTR`) represents the time it takes to repair a service that has failed. Again, smaller values are better than larger ones. Finally, the dimension called `availability` represents the probability that a service is available. For the `availability` dimension, larger values are better than smaller values.

In Figure 4 we also define a contract called `systemReliabilty` of type `Reliability`. The contract specifies constraints over the dimensions defined in the

```
interface RateServiceI {
  Rates latest(in Currency c1,in Currency c2)
    raises(InvalidC);
  Forecast analysis(in Currency c) raises(Failed);
};
```

FIG. 3. The `RateServiceI` interface described in CORBA IDL

```
type Reliability = contract {
  numberOfFailures: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  availability: increasing numeric;
};

type Performance = contract {
  delay: decreasing numeric msec;
  throughput: increasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no / year;
  TTR {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3
      };
  availability > 0.8;
};

rateServerProfile for RateServiceI = profile {
  require systemReliability;
  from latest require Performance contract {
    delay {
        percentile 80 < 20 msec;
        percentile 100 < 40 msec;
        mean < 15 msec
        };
  };

  from analysis require Performance contract {
    delay < 4000 msec
  };
};
```

FIG. 4. Contracts and Profile for RateServiceI

`Reliability` contract type. The first constraint specifies an upper bound for the number of failures. The second constraint applies to the `TTR` dimension. This constraint uses statistical properties, such as mean, variance, and percentiles, to characterize QoS along the `TTR` dimension. In QML, we refer to such statistical properties as dimension *aspects*. The aspect "**percentile** 100 < 2000" states that the 100th percentile must be less than 2000.

The profile `rateServerProfile` associates contracts with entities in the `rateServiceI` interface. The first requirement clause in the profile states that the service should satisfy the previously defined `systemReliability` contract. Since the clause does not refer to any particular operation, it is considered a default requirement that applies to every operation within the `rateServiceI` interface. Being part of a default requirement, the `systemReliability` contract is called a *default contract* for the profile. Contracts for individual

operations are allowed only to strengthen (refine) the default contract. In the `rateServerProfile` there is no default performance contract; instead we associate individual performance contracts with the two operations of the `RateServiceI` interface. For `latest` we specify in detail the distribution of delays in percentiles, as well as a upper bound on the mean delay. For `analysis` we specify only an upper bound and can therefore use a slightly simpler syntactic construction for the expression. Since throughput is omitted for both operations, there are no requirements or guarantees with respect to this dimension.

We have now specified example reliability and performance requirements for the `rateServiceI` interface. Although the `rateServerProfile` is specified in terms of an interface (`rateServiceI`), it characterizes the QoS of a particular implementation of this interface. We can specify multiple profiles for the same interface, and use distinct profiles for different implementations. The key to this flexibility is that QoS specifications are not embedded within an interface, but defined as separate entities.

Intuitively we would say that the constraint "`delay` < 10" is stronger than the constraint "`delay` < 20." This relationship between the two constraints is due to the fact that `delay` is a decreasing dimension (smaller values are better) and the fact that the value 10 is smaller than the value 20. In QML, we formalize this notion of "stronger than" for constraints and define a general conformance relation over constraints. Stronger constraints conform to weaker constraints. We then use this conformance relation on constraints to define conformance relations on contracts and profiles. Conformance is an important aspect of QoS specifications because it enables us to compare specifications based on constraint satisfaction rather than exact match. As we show in Section 4, conformance is essential for implementing a QoS-based trader: the QoS-based trader should select any service whose QoS properties conform to the client's requirements, the trader should not just select the services whose properties are identical to the client's requirements.

QoS specifications can be used in many different situations. They can be used during the design of a system to understand and document the QoS requirements that must be imposed on individual components to enable the system as a whole to meet its QoS goals. In [6] we show how to use QML at design time. The focus of this paper is the use of QoS specifications as first-class entities at runtime.

3. QRR: A QML-Based QoS Fabric

Our QoS fabric, QRR, is based on the following requirements:

1. QRR should support the same fundamental concepts as QML. We want to use the same QoS specification concepts during design and implementation. Using the same concepts implies that QML's precise, formal definition [5] carries over to QRR. A precise definition improves the interoperability of different QRR/QML components.
2. Since some QoS requirements may not be known until runtime, it should be possible to dynamically create new QRR specifications. Rather than use dynamic compilation for such specifications, we want to call generic creation functions in the QRR library.
3. It should be possible to explicitly check consistency of dynamically created specifications against the static semantic rules of QML/QRR. The QML compiler checks the rules for compiled specifications. We need a library function that checks the rules for dynamically created specifications.
4. Once created, there should be ways to manipulate QRR specifications. For example, a QoS offering by a server may have to be adjusted relative to the current execution environment to accurately reflect what QoS the client will actually receive.
5. QRR should impose a minimal overhead and be scalable.
6. QRR should provide a minimal set of generic building blocks for runtime QoS specification. In particular, QRR specifications should be independent of the mechanisms and applications that use them. For example, in negotiation, as well as trading, we are interested in agreements between parties involving commitments from both sides. Thus we are dealing with structures consisting of pairs of QoS specifications (one for each party). Rather than provide an agreement abstraction in QRR, we only provide the basic building blocks that represent QoS specifications. It is then up to the mechanisms and applications to use these basic building blocks to create composite structures.

We are implementing QRR to satisfy these requirements. Currently, we have implemented a prototype QML compiler and a prototype QRR library. We have successfully compiled QML specifications into QRR, instantiated those specifications in a CORBA environment, communicated the specifications between distributed components, and compared them using a conformance checking function that is part of the QRR library.

3.1 *Implementing QRR*

The QRR implementation contains a generic C++ library that allows applications to create QRR specifications and to check conformance of these specifications. This library is linked into applications that use QRR. The library defines a number of data types that are used to represent QRR specifications in C++. These data

types are generated from CORBA IDL type definitions to facilitate the communication of QRR specifications between distributed CORBA objects.

In addition to the generic library, the implementation also contains a QML to QRR compiler. This compiler emits a mix of IDL and C++ code to represent a particular QML specification. The emitted IDL code consists of types that represent that QML specification. The C++ code contains functions to create QRR instances of the QML specification. The emitted IDL code is translated into C++ using a conventional CORBA IDL compiler.

3.2 Representation

We describe how to represent QML constructs in terms of CORBA IDL and C++. Profiles are represented as instances of the `profile` struct shown in Figure 5. They contain the profile name, the interface name, a sequence of default contracts (`dcontracts`), and a sequence (`profs`) of structs, each associating an interface entity with a set of contracts. The `profs` sequence represents the individual contracts of the profile. In QRR, all profiles are instances of the `profile` struct. For a particular profile specified in QML, the QML compiler emits a C++ function that constructs an instance of the `profile` struct. The C++ function also constructs and assigns appropriate data structures to the fields of this struct.

```
struct profile {
   string pname;
   string iname;
   contractSeq dcontracts;
   entityProfileSeq profs;
};
```

FIG. 5. IDL for profile

QoS constraints are represented as instances of the struct `constraint` in Figure 7. A `constraint` struct has a sequence of `aspect` structs, as well as a tag indicating whether it is a simple constraint—such as "*delay* < 10"—or a set of aspects representing statistical characterizations. We define a separate struct type for each aspect kind, however, the figure only shows the struct used to represent mean aspects. Because IDL does not allow polymorphism for structs, we cannot directly reflect the relationship between the general notion of aspects, captured by the `aspect` struct, and particular aspect types such as `mean`. Instead of defining particular aspect types as subtypes of `aspect`, we define an *any* field in instances of `aspect` that contains a particular aspect instance. We also define a type tag in instances of `aspect` that indicates which particular aspect type has been wrapped in the *any* field.

We provide two alternative representations for contracts and contract types. In the *generic* representation, all contracts are instances of the same type, and this type is then part of the QRR library. In the *static* representation, only contracts of the same QML contract type are instances of the same QRR type. In addition, the QRR types used for the static representation are emitted by the QML compiler.

The static representation requires that the emitted QRR types are linked into the application that instantiates them. On the other hand, the static representation facilitates a more efficient implementation of conformance checking and other QRR functions.

With the generic representation, applications can dynamically create and communicate contracts whose types are not known at compile time. However, manipulation and analysis of contracts is less efficient in the generic representation because the structure of contracts must be discovered dynamically.

Although we describe them as separate representations, our goal is to allow their simultaneous use to achieve maximum flexibility and performance. Since our current implementation only supports the static representation, we only give a brief overview of the generic representation and concentrate primarily on the static representation.

```
enum aspectKind {
   ak_freq, ak_perc,
   ak_mean, ak_var,
   ak_simple
};

struct mean {
   operators op;
   value num;
};

struct aspect {
   aspectKind ak;
   any asp;
};

typedef sequence <aspect> aspects;

enum constrKind { ck_simple, ck_stat };

struct constraint {
   constrKind ck;
   aspects asps;
};
```

FIG. 7. IDL for aspect and constraint

The IDL definitions in Figure 8 describe some elements of the generic contract representation. In the generic representation, all contracts are instances of the struct called `contract`. A contract's dimensions are then represented as a sequence of structs of type `dimension`. A contract has a type identifier (`tid`) that refers to its contract type. Contract types are built from various generic structs that capture all the information about domains and their ordering. These type representations are quite elaborate as they contain information about all values, how these values are ordered, and whether the dimension is increasing or decreasing. Due to space constraints we do not describe these type structures in detail in this paper.

With the static representation, the QRR compiler will map a QML contract type into a number of C++ classes and an IDL struct. The C++ classes represent the contract type itself. They contain information about the domain elements and the domain ordering for the dimensions defined in the contract type. The IDL struct is used to represent contracts that are instances of the contract type. An instance of the emitted IDL struct represents a particular contract.

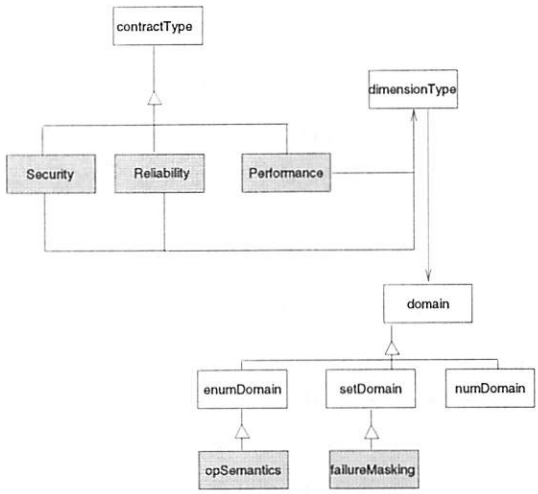


FIG. 6. Class diagram for contract type representation.

```
struct dimension {
   string name;
   constraint constr;
};

struct contract {
   tid ct;
   sequence<dimension> dims;
};
```

FIG. 8. IDL for generic contracts

The emitted C++ classes inherit from, and adds to, a set of contract type base classes implemented in the QRR library. Figure 6 shows—using UML [3] notation—a simplified view of the C++ classes for contract types in the static representation. Emitted classes are grayed and classes defined in the QRR library are white.

Sub-classes of `contractType` represent the emitted classes for specific contract types. These classes contain data members that represent the dimensions in the contract type. They also contain a conformance checking function. Since the conformance checking function is emitted on a per-contract type basis, it can directly refer to the type's dimensions as data members.

If the contract types contain set or enumeration domains that are ordered, we also emit C++ classes that represent these domains. The domain classes are subclasses of the library class called `domain`. The main role of emitted domain classes is to provide information about the domain ordering.

In addition to the C++ classes, the compiler also emits an IDL struct definition for each contract type. The name of this struct is the contract type name with *_i* appended to it. The struct has one field for each dimension. Each field has the same name as the corresponding dimension and is of type `constraint`.

In Figure 10 we show a QML contract type called `Reliability` and the corresponding emitted IDL struct.

An instance of the `Reliability_i` struct will hold instances of constraints that in turn hold the aspects specified for each individual constraint. An instance also contains the type identifier of its contract type.

Given a type definition, such as `Reliability_i`, the programmer could in principle create contracts at runtime by instantiating the type and building up appropriate structures for the fields in the struct. However, this is tedious. To automate the instantiation process, the QML compiler emits instantiation functions for each contract and profile declared in QML.

To give a concrete, but simple, example of how these instantiation functions are constructed, we provide a contract in Figure 11 and the corresponding emitted construction function in Figure 13. Running these simple definitions through the QML to QRR compiler will produce the static representation which is a struct with name `T_i`. The compiler also emits the C++ class `T`, which describes the contract type and implements conformance checking. In addition, it produces a function—shown in Figure 13—with the same name as the specified contract (in this case `C`). When `C` is called it will return an instance of `T_i` representing the constraints specified in `C`. The `C` function uses the same lower level functions as are provided to applications that manually composes contracts and profiles. Similarly, we produce functions for profiles that build up the corresponding QRR structures.

3.3 Library Functions

When an application needs to check conformance, it invokes the library function `conformsTo` whose signature is shown in Figure 12. This function takes two profiles, and checks conformance between their contracts. Inside profiles, contracts are stored as a pair consisting of a contract type name and an element of type *any*. For a performance contract, the *any* element will contain an instance of type `Performance_i` and the contract type name will be the string "Performance". To check conformance between performance contracts, the `conformsTo` will use the string "Performance" to lookup the C++ object which represents performance contract types at runtime. This object is of type `Performance` and will have a virtual function called `conformsTo` (the signature of this function is given in Figure 12 as `Performance::conformsTo`). The `Performance::conformsTo` function is emitted. It expects two *any* arguments that both contain instances of the struct `Performance_i`. Since it is emitted, the `Performance::conformsTo` function knows which objects to extract from the *any* arguments.

```
type Reliability = contract {
  numberOfFailures: decreasing numeric;
  TTR: decreasing numeric;
  availability: increasing numeric;
};

struct Reliability_i {
   tid ct;
   numberOfFailures constraint;
   TTR constraint;
   availability constraint;
};
```

FIG. 10. IDL for statically generated contracts

```
type T = contract {
   l : increasing numeric msec;
   s : enum {initial, amnesia,
         noguarantee, rolledback};
};

C = T contract {
   l {
      percentile 40 < 50 ;
      mean < 20
     };
   s == amnesia;
};
```

FIG. 11. A simple contract type and contract

```
int conformsTo(profile &stronger,profile &weaker);

int Performance::conformsTo(CORBA::Any *stronger,
                            CORBA::Any *weaker);

int checkSem(const profile &p);
```

FIG. 12. Some library function signatures

The programming model also provides a variety of convenience functions for creating aspects, contracts and other QRR/QML constructs.

3.4 Programming Model

To give the reader a better feel for the programming model offered by QRR, we describe a simple QoS compatibility-checking mechanism that allows a client to send its QoS requirements in the form of a QRR profile to a server. The server checks whether it can satisfy the client's requirements.

```
interface QoSAware {
   exception invalidProfile{};
   boolean compatible(in profile p)
       raises (invalidProfile);
};
```

FIG. 9. QoSAware interface

To support the QoS-checking mechanism, the server implements the interface `QoSAware`, which we describe

```
T_i * C(){
   T_i * _C; _C = new T_i;
   _C->ct =  CORBA::string_dup("T");
   //Create aspects for l
   aspect * _l;
   _C->l.asps.length(2);
   _C->l.ck = ck_stat;
   _l = qml_perc_asp(le,40,(float)50);
   _C->l.asps[0] = *_l;
   delete _l;
   _l = qml_mean_asp(le,(float)20);
   _C->l.asps[1] = *_l;
   delete _l;
   //Create simple for s;
   aspect * _s;
   _C->s.asps.length(1);
   _C->s.ck = ck_simple;
   _s = qml_simp_constr(eq,(float)2/*amnesia*/);
   _C->s.asps[0] = *_s;
   delete _s;
   return _C;
};
```

FIG. 13. Emitted function that creates a T contract

```
CORBA::Environment env;
profile * p = i2_prof();
if (I2ref->compatible(*p,env) {
   //OK to use this server
....
} else {
   //use another server
....
};
```

FIG. 14. Client call

in Figure 9. The operation `compatible` allows the client to send the profile it requires to the server. The server responds with true if the client's requirements and the server's capabilities are compatible; and with false otherwise. If the profile is semantically invalid, the operation raises an exception.

To make the QoS checking more concrete, let us assume that a server A provides an interface I_1 and uses a server B that implements an interface I_2. We can describe, in QML, the requirements of server A on server B as a profile for the interface I_2. We can also describe the QoS provided by A as a profile for interface I_1. Having defined those profiles and the contracts that they use we can emit QRR code that can be compiled and linked with both servers. Notice that server A plays the role of client relative to server B.

We can create the specified profiles in server A by invoking the emitted functions that have the same names as the profiles specified in QML. If we have a profile named `i2_prof` specifying A's requirements on I_2, A objects would use an emitted function called `i2_prof` to create an QRR instance of this profile. The C++ code in Figure 14 illustrates how a profile can be created and sent with an ordinary CORBA request.

```
CORBA::Boolean B_serverImpl::compatible(
                    const profile &p,
                    CORBA::Environment &_ev)
{
   if (! checkSem(p1) {
      throw QoSAware::invalidProfile();
   };

   if(conformsTo(myprof(),p1)){
      cout << "Conformance... " << endl;
      return 1;
   }
   else {
      cout << "Non-conformance..." << endl;
      return 0;
   }
};
```

FIG. 15. Server implementation

The implementation of `compatible` simply takes the profile specified for the server and checks its conformance to the profile supplied by the client. We assume that the server obtains its own profile by invoking a function called `myprof`. The implementation of `compatible` checks the static semantics of the profile before doing performance checking. In the future we intend to include information in a profile that allows a program to determine whether a profile has already been checked for semantic validity or not. With this extra information, we can avoid redundant semantic checks. Figure 15 describes a simple implementation of a server that supports the `QoSAware` interface.

3.5 Discussion

In the generic mapping, contracts can be created, embedded in profiles, and communicated even if they are not statically known. If we use this mapping, we would initially send contract type descriptions for all contract types to be used during a session. This would ensure that each participating object has all contract type descriptions available. Using the static mapping, on the other hand, we require that all contract types are known statically and compiled into the participating objects. If a received contract is an instance of an unknown type, the receiver can do little more but raise an exception.

We could require that it is decided up front whether the generic or static mapping will be used during a session. Having a strict separation of the generic and static mapping, and only use one of them during a particular session, would simplify the implementation, but it would also be quite inflexible. In the case of the generic mapping, it also imposes unnecessary performance overhead when contract types are already known.

Global consistency of types is another issue that we are considering. It is necessary to have a mechanism for identifying and comparing types to determine if they are the same. The current implementation uses an oversimplifying approach based on text strings. Our plans for future work includes leverage from previous solutions such as the handling of types in CORBA to resolve this issue in QRR.

4. Example: A Qos-Based Trader

This section illustrates how QRR can be used to construct higher-level QoS components. We show how to build a QoS-based trader, and explain its utility in the context of the currency trading example from the introduction section (do not confuse a QoS-based trader with the currency trading service in the currency trading system).

The purpose of this section is not to discuss or advocate the merits of QoS-based trading, but to show con-

cretely the expressive power and flexibility of QRR. The constructs of QRR make it relatively straightforward to implement QoS-aware components that would otherwise be complicated to build. For another example, see [10] in which a QoS negotiation mechanism is presented. The mechanism uses QML and QRR as its underlying mechanisms for exchange of QoS specifications.

4.1 Trading in Distributed Systems

A conventional trader [15] in distributed systems facilitates the binding between clients and services. Services register with the trader, and clients query the trader to find a service that satisfies certain criteria. We show an example interface to a very simple conventional trader in Figure 16.

Services register themselves by calling the `offer` method on the trader. A service passes a description of its properties—the properties that can be used for service selection—and a reference to itself. The trader returns an offer identifier to the service. The service can use this identifier to later withdraw its offer by invoking the `withdraw` method. Clients call the methods `find` and `findAll` to obtain service references. The `find` method returns a single service reference. The found service satisfies the criteria passed in as parameter to the `find` call. The `findAll` method returns a list of service references; it returns all the services that match the criteria passed in as parameter.

In general, a trader matches criteria passed in by clients against properties passed in by services. Conventional trader services typically use name-value pairs for server selection. Services pass in one such attribute list when they register, and clients pass in an attribute list that is matched against such service attribute lists. Typically, matching is based on name-value equality, with the provision that the client's attribute list must be equal to a sub-list of the server's attribute list.

4.2 QoS-Based Trading

The main idea behind QoS-based trading is to use QoS specifications as service properties and client criteria, and to use specification conformance to perform the criteria-to-properties matching. We show that with QRR it is relatively straightforward to implement a QoS-based trader.

```
interface Trader {
  OfferId offer(in ServiceProperties sp,
    in Object obj) raises (invalidOffer);
  Match find(in Criteria cr) raises(noMatch);
  MatchSeq findAll(in Criteria cr) raises(noMatch);
  void withdraw(in OfferId o) raises(noMatch);
};
```

FIG. 16. The interface of a simple conventional trader

We want to use the QoS-based trader to establish QoS agreements between clients and services in distributed object systems. A QoS agreement is a contract between a client and a service. In our discussion so far, we have talked about client requirements and service properties. This is a somewhat simplified picture. The service properties may depend on the way in which the client uses the service. For example, the throughput that a service can provide may depend on how frequently a client calls the service. So, for performance, a QoS contract may involve requirements and properties for both clients and services.

Thus in general, the `ServiceProperties` argument to the `offer` method in a QoS-based trader will involve service properties and requirements. The service can provide its properties if the client satisfies the requirements. Figure 17 gives a possible structure for `ServiceProperties` using QRR. We describe the service requirements and properties using profiles. Figure 17 also shows the structure of client criteria. These also have a two profiles: one representing client requirements and one representing client properties.

With the `conformsTo` function on profiles introduced in Section 3.1, we can now implement the matching procedure in the QoS-based trader. We illustrate the conformance-based matching procedure in Figure 18. The procedure iterates through the registered services and for each service checks whether that service satisfies the criteria passed in as arguments. The function `conformsTo` takes two profiles and determines whether the first profile conforms to the second profile. To find a matching service, the `find` method checks whether the server properties conform to the client requirements, and it checks whether the client properties conform to the server requirements.

An alternative implementation strategy would be to use a conventional trader and represent QoS specifications as name-value pairs. However, we would then be limited by the expressive power of name value pairs; it is not clear how to elegantly represent the concepts of QML and QRR in terms of name-value pairs. More seriously, with a conventional trader we would use equality for server selection. It is essential that we can select a service even if the client's requirements are not equal to the service's properties: we want to select a service as long as the service's properties satisfies, or *conforms to*, the client's requirements.

4.3 Using a QoS-Based Trader

Here, we use a QoS-based trader to implement the currency trading system from the introduction section. Currency trading is a complex process that requires sig-

```
struct ServiceProperties {
  profile properties;
  profile requirements;
};

struct Criteria {
  profile properties;
  profile requirements;
}
```

FIG. 17. The structure of `ServiceProperties` and `Criteria`

nificant information and analysis [13]. Although we appreciate the complexity of systems supporting such trades, we have to simplify the problem for the purpose of this paper.

As a reminder, our simple currency trading system consists of three logical components: a front-end that serves as a user interface, a rate service that provides information about current exchange rates, and a currency trading service to execute currency trades. We will focus only on the rate service that provides exchange rate information. The structure of this simple currency trading systems was shown in Figure 2.

Assume that we want to build a currency trading system that uses a rate service available on the Internet. Different service providers may implement different rate services with the same basic functionality, but with different QoS properties. For example, one rate service may provide frequent exchange rate updates and thus higher precision. Another rate service may provide less frequent update, which implies that the information will not be as accurate. Different services may provide information for different currencies. Moreover, one rate service may be highly available and expensive to use, whereas another rate service may be more unreliable but cheaper

```
// C++ Implementation sketch of the find method
// in a QoS-based trader
Match QoS-Trader::find(const Criteria &cr)
 throw(noMatch)
{
 ServiceIterator it = ...;
 for(it.init(); ! it.done(); it.advance())
   serviceProperties sp =
     it.currentServiceProperties();
   if(conformsTo(sp.properties,cr.requirements) &&
      conformsTo(sp.requirements,cr.properties)){
        return it.currentServiceMatch();
      }

 throw noMatch();
};
```

FIG. 18. Matching based on conformance

to use. The point is that one size does not fit all: different clients have different requirements or expectations about the QoS delivered by a rate service, and different clients are willing to pay for high levels of QoS, whereas other clients are not.

The QoS-based trader provides a mechanism for performing server selection in this kind of environment. The various rate services register themselves with the QoS-based trader and provide QoS specifications that reflect their particular notion of QoS. Clients then consult the QoS-based trader when they want to connect to a rate service. In doing so, clients communicate their QoS expectations to the QoS-based trader to select a suitable service.

Figure 19 shows the structure of the currency trading system with a QoS-based trader. We could of course select between multiple currency trading services as well, but we want to simplify the example and focus on rate

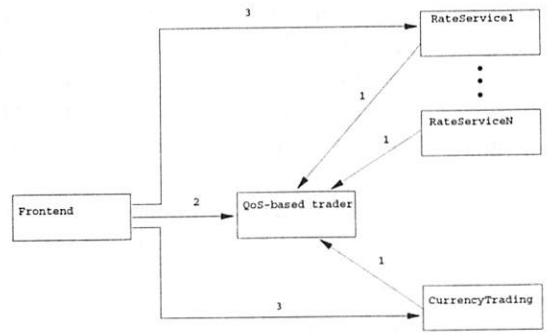


FIG. 19. Structure of the currency trading system with a QoS-based trader

```
type DataQuality = contract {
  currencies: set { USD, JPY, SEK, FIM , DKK, DEM
                ITL, KGS, EEK, KYD, BWP, LUF, FRF,
                GBP, QAR,  RUR, TOP, MAD, SAR };
  updateFrequency: increasing numeric updates/min;
};

type Reliability = contract {
  availability: increasing numeric;
  referenceValidity = increasing
    enum { invalid, valid }
    with order { invalid < valid };
  ...
}

type ClientPriceBound = contract {
  costPerInvocation: decresing numeric cent/call;
  costPerHour: decreasing numeric cent/hour;
}
```

FIG. 20. Contract types for the currency trading system

services. The issues involved in selecting a currency trading service are similar.

In the following we show how to use the concepts of QML and QRR to create QoS specifications for a couple of rate services and for a front-end. Due to space constraints, we provide a somewhat simplified version of the various QoS specifications.

First, we need a number of contract types to represent the various QoS categories under consideration in the QoS agreement between the front-end and a rate service. Figure 20 outlines these contract types. The first type, called `DataQuality` captures the notion of data quality: accuracy and content. The `currencies` dimension reflects the currencies that a particular service can provide information on. The `updateFrequency` reflects how often this rate information is updated at the server and thus gives an indication of the precision. The `Reliability` contract type captures the reliability of the rate service. In our previous work, we identified a number of dimensions to characterize reliability for distributed object systems [9]. Here, we only use a very simple characterization. We use two dimensions: availability is the probability that the server is up when trying to contact it and reference validity states whether the object reference to the server remains valid after the server has crashed and come back up. Finally, the `ClientPriceBound` contract type captures how much the front-end is willing to pay for the rate service. Specifications of this type can characterize an upper bound, or the exact price, for the service cost as seen by the client. An upper bound does not determine the actual price the service charges, it merely serves as a first-order filtering criterion when matching up clients and services. The actual determination of how to charge for a service may involve more sophisticated negotiation protocols that we do not consider here. Payment can be specified both as a per-invocation cost and a per-session cost, where the per-session cost is determined from the length of the session.

```
service1props for rateService = profile {
  require DataQuality contract {
    currency >= { USD, JPY, SEK, FIM , DKK, DEM
                  ITL, KGS, EEK, KYD, BWP, LUF, FRF,
                  GBP, QAR,  RUR, TOP, MAD, SAR };
    updateFrequency >= 1;
  };
  require Reliability contract {
    availability >= 0.99;
    referenceValidity == valid;
  };
  require Price contract {
    costPerInvocation <= 100;
  };
};

service2props for rateService = profile {
  require DataQuality contract {
    currencies == { SEK, FIM, DKK, MAD, GBP };
    updateFrequency >= 60;
  };
  require Price contract {
    costPerInvocation <= 50;
  };
};
```

FIG. 21. Profiles for rate services

Given these basic contract types, we can now start to define the QoS properties of services. To simplify the example, we only specify service properties and client requirements. In other words, we do not consider client properties and server requirements. Moreover, we specify the contracts as default contracts—contracts that apply to the rate service object rather than individual methods in this object.

In Figure 21, we outline the QoS properties for two different rate services. Notice that both services implement the `RateService` interface defined in Figure 3. The first service is characterized by the `service1props` profile. This service is a highly available, expensive service that supports many currencies with a low update frequency. The second service is characterized by the `service2props` profile. The second service does not provide any reliability guarantees. On the other hand it is fairly cheap to use. It supports only a few currencies, but the frequency of updates is high. Both services can only charge based on a per-invocation scheme, they cannot charge for entire sessions.

We can now create profiles, using QRR, during server initialization. One service will create a profile according to the `service1props` specification, and the other service will create a profile according to the `service2props` specification. We can create the QRR profiles by calling the profile-creation functions emitted by the QRR compiler. Alternatively, we can bypass the QRR compiler and create the profiles by calling the QRR library functions directly. For simplicity we assume that each service has a single profile. In a more realistic situation, each service may have multiple profiles to handle service differentiation and QoS variations over time.

Both services will register with the QoS-based trader, and when registering, they will provide their respective profiles. In a dynamic environment services can withdraw old offers that can no longer be supported and make new offers.

We also need to specify the requirements of the front-end. We give an example of such a specification in Figure 22. According to the figure, the front-end has no reliability requirements, it is willing to pay medium cost on a per-invocation basis, it requires rates for only Swedish Crowns, Danish Crowns, and British Pounds, and it re-

quires relatively high update frequency. This specification can either reflect the requirements of the front-end object as such, or it can represent the requirements of a user of the front-end object. In the first case, we can write the profile in QML and generate profile-creation functions based on the QML specification—we know the QoS requirements when we implement the front-end. In the second case, where the requirements reflect user requirements, we do not know the requirements until runtime, and we cannot compile profile creation functions into the front-end. In this case, we have to call the generic profile creation functions in QRR to dynamically create a profile that reflects the user's requirements.

Once the front-end has created a profile that reflects its requirements, it can then call `find` on the QoS-based trader to obtain a reference to a rate service object that satisfies those requirements. In our case, the front-end's requirements will give rise to selection of service number two. The profile `service2props` conform to the profile `frontendReqs`, where as the profile `service1props` does not.

4.4 Discussion

A QoS-based trader facilitates the server-selection process in open systems where clients are not built to work with one particular server. In an open system, clients must be prepared to select from a range of different services that provide the same functionality at different levels of QoS. The QRR fabric makes it relatively straightforward to implement a QoS-based trader. In contrast, it is non-trivial to implement a similar functionality using a conventional trader with name-value pairs.

The QoS-based trader that we presented here does not solve the whole issue of establishing QoS agreements between clients and services. We ignored the issue of agreement duration. Furthermore, we only touched on the topic of payment for QoS. In the example, we described a very simple way to perform a first-order screening based on how much clients are willing to pay. What clients actually pay may be less than this upper bound, and will probably be the result of further negotiation between the client and the server given that the upper bound is satisfied. Finally, the trader does not address the issue that services may provide different levels of QoS depending on the dynamic environment. For example, a service may be able to provide higher levels of QoS in an environment with plentiful resources and few clients. We believe that these issues can be addressed as extensions to the simple QoS-based trader that we described. The issues will not change the basic functionality of the trader. Many of the issues can possibly be addressed as separate QoS components that complement the trader.

```
frontEndReqs for rateService = profile {
  require DataQuality contract {
    currency == { SEK, DKK, GBP};
    updateFrequency >= 10;
  };
  require Price contract {
    costPerInvocation <= 75;
  };
}
```

FIG. 22. Front-end profile

5. Related Work

Generally, interface definition languages, such as OMG IDL [14], specify functional properties, but lack any notion of QoS.

TINA ODL [20] is different in that it allows programmers to associate QoS requirements with streams and operations. A major difference between TINA ODL and our approach is that they syntactically include QoS requirements within interface definitions. Thus, in TINA ODL, one cannot associate different QoS properties with different implementations of the same functional interface.

Similarly, Becker and Geihs [1] extend CORBA IDL with constructs for QoS characterizations. Their approach suffers from the same problem as TINA ODL: they statically bind QoS characterizations to interface definitions. They also allow QoS characteristics to be associated only with interfaces, not individual operations. In addition, they support only limited domains and do not allow enumerations or sets. Finally, they allow inheritance between QoS specifications, but it is unclear what constraints they enforce to ensure conformance. QoS specifications are exchanged as instantiations of IDL types without any particular structure.

There are a number of languages that support QoS specification within a single QoS category. The SDL language [11] has been extended to include specification of temporal aspects. The RTSynchronizer programming construct allows modular specification of real-time properties [17]. In [7], a constraint logic formalism is used to specify real-time constraints. These languages are all tied to one particular QoS category, namely timing. In contrast, QML and QRR are general purpose; QoS categories are user-defined types in QML, and can be used to specify QoS properties within arbitrary categories.

The specification and implementation of QoS constraints have received a great deal of attention within the domain of multimedia systems. In [18], QoS constraints are given as separate specifications in the form of entities called QoS Synchronizers. A QoS Synchronizer is a distinct entity that implements QoS constraints for a group of objects. The use of QoS Synchronizers assumes

that QoS constraints can be implemented by delaying, reordering, or deleting the messages sent between objects in the group. In contrast to QML, QoS Synchronizers not only specify the QoS constraints, they also enforce them. The approach in [19] is to develop specifications of multimedia systems based on the separation of content, view, and quality. The specifications are expressed in *Z*. The specifications are not executable per se, but they can be used to derive implementations. In [2], multimedia QoS constraints are described using a temporal, real-time logic, called QTL. The use of a temporal logic assumes that QoS constraints can be expressed in terms of the relative or absolute timing of events. Campbell [4] proposes pre-defined C-language structs that can be instantiated as QoS specifications for multimedia streams. The expressiveness of the specifications are limited by the C language, thus there is no support for statistical distributions. Campbell does, however, introduce separate attributes for capturing statistical guarantees. It should be noted that Campbell does not claim to address the general specification problem. In fact, he identifies the need for more expressive specification mechanisms that include statistical characterizations. In contrast to QML, the multimedia-specific approaches only address QoS within a single domain (multimedia). Moreover, these approaches tend to assume stream-based communication rather than method invocation.

Zinky et al. [23, 22] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The notion of a *connection* between a client and a server is a fundamental concept in their framework. A connection is essentially a QoS-aware communication channel; the expected and measured QoS behaviors of a connection are characterized through a number of *QoS regions*. A region is a predicate over measurable connection quantities, such as latency and throughput. There approach does not seem to enable dynamic creation, communication, and manipulation of QoS specifications. In particular, it is not clear how to use their approach to dynamically establish connections in an open environment based on QoS needs and provisions.

In [21] Zinky and Bakken discuss the problem of managing meta-information in systems with adaptable QoS. The paper discusses various kinds of data that is needed for adaptive CORBA systems. They do not, however, present any concrete way in which the information can be described and communicated. We believe that QML and QRR can be used to described several of the facets—such as *kind* and *mechanism*—identified in the paper.

Linnhoff-Popien and Thissen [12] describe methods for evaluating the distance from the ideal characteristics of a required service to what the available offers provide. They use computed distances to select the most appropriate service. In contrast, QML and QRR focuses on statistical characterization of QoS and systematic comparison by means of conformance. We could extend our approcah to incorporate a notion of "preference" if many services satisfy a client's requirements. The area of utility theory is a promising foundation for such an extension.

Within the Object Management Group (OMG) there is an ongoing effort to specify what is required to extend CORBA [14] to support QoS-enabled applications. The current status of the OMG QoS effort is described in [16], which presents a set of questions on QoS specification and interfaces. We believe that our approach provides an effective answer to some of these questions. ISO has an ongoing activity aiming at the definition of a reference model for QoS in open distributed systems. In a recent working paper [8] they outline how various dimensions such as delay and reliability could be characterized. They lack, however, any proposal or recommendations for representations or languages with which such constraints can be expressed and communicated.

6. Conclusion

We believe that one of the next major advances of distributed object systems is to make them QoS enabled. An important step towards QoS enabling is to facilitate QoS characterizations of distributed object components. We have previously suggested the QML language for this purpose [6]. When we have characterizations we need to allow these to influence how distributed objects are connected and what underlying communication mechanism and transports they use. Currently, these decisions are typically made at design time and hardwired into the system. However, to build flexible applications that execute in internet-like environments, we need to support dynamic connections based on QoS matching. The dynamic connections can be facilitated by QoS components, such as traders and negotiators. A QoS characterizations is of little use if we can not verify that components of a system actually complies to the QoS agreements that have been set up among them. This can be accomplished by monitoring connections between objects.

Trading, negotiation, monitoring and many other functions of QoS-enabled distributed systems *require* a format for exchanging QoS specifications. We have designed and are implementing a language (QML) and a run-time representation (QRR) that is tailored for making distributed object systems QoS aware. One of the main advantages of using QRR instead of ad-hoc runtime representations is that QRR comes with a precisely defined notion of conformance. Moreover, the QRR library comes with a generic conformance checking function. Although conformance appears somewhat manageable for simple constraints over numeric dimensions, it is chal-

lenging to define and check conformance for statistical aspects and set domains with user-defined ordering.

QML and QRR allow middle-ware developers to invent new mechanisms and services for QoS-enabled distributed systems.

OMG has standardized—among other things—CORBA IDL and IIOP to facilitate interoperability of heterogeneous distributed objects. Analogously, we believe that open systems can only meet QoS requirements if they can specify and communicate their QoS characteristics and requirements. QML and QRR could be viewed as a first attempt to come up with a common specification language and inter-change format for QoS enabled distributed object systems.

References

1. C. R. Becker and K. Geihs. Maqs—management for adaptive qos-enabled services. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.
2. G. Blair, L. Blair, and J. B. Stefani. A specification architecture for multimedia systems in open distributed processing. *Computer Networks and ISDN Systems*, 29, 1997. Special Issue on Specification Architecture.
3. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language*. Rational Software Corporation, January 1997. version 1.0.
4. A. T. Campbell. *A Quality of Service Architecture*. PhD thesis, Lancaster University, January 1996.
5. S. Frølund and J. Koistinen. QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories, February 1998.
6. S. Frølund and J. Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, April 1998.
7. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of Real-Time Systems Symposium*. IEEE, December 1997.
8. ISO. Working draft for open distributed processing—reference model—quality of service. Result from the SC21/WG7 Meeting, July 1997.
9. J. Koistinen. Dimensions for reliability contracts in distributed object systems. Technical Report HPL-97-119, Hewlett-Packard Laboratories, October 1997.
10. J. Koistinen and A. Seetharaman. Worth-based multi-category quality-of-service negotiation in distributed object infrastructures. In *Proceedings of 2nd International Enterprise Distributed Object Computing Workshop (EDOC'98)*, November 1998.
11. S. Leue. Specifying real-time requirements for sdl specifications—a temporal logic-based approach. In *Proceedings of the Fifteenth IFIP WG6 (Protocol Specification, Testing, and Verification XV)*, June 1995.
12. C. Linnhoff-Popien and D. Thissen. Integrating qos restrictions into the process of service selection. In *Proceedings of the Fifth IFIP International Workshop on Quality of Service*, May 1997.
13. C. Luca. *Trading in the Global Currency Markets*. Prentice-Hall, 1995.
14. Object Management Group. *The Common Object Request Broker: architecture and specification*, revision 2.0 edition, July 1995.
15. Object Management Group. *CORBA Services—Trader Service*, formal/97-07-26 edition, July 1997.
16. Object Management Group. *Quality of Service: OMG Green paper*, draft revision 0.4a edition, June 1997.
17. S. Ren and G. Agha. Rtsynchronizer: Language support for real-time specifications in distributed systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*. ACM, June 1995.
18. S. Ren, N. Venkatasubramanian, and G. Agha. Formalizing multimedia qos constraints using actors. In *Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems*, 1997.
19. R. Staehlin, J. Walpole, and D. Maier. Quality of service specification for multimedia presentations. *Multimedia Systems*, 3(5/6), November 1995.
20. Telecommunications Information Networking Consortium. *TINA Object Definition Language*, June 1995.
21. J. A. Zinky and D. E. Bakken. Managing systematic meta-data for creating qos-adaptive corba applications. In *Proceedings of the Fifth IFIP International Workshop on Quality of Service*, May 1997. Short paper.
22. J. A. Zinky, D. E. Bakken, and R. D. Schantz. Overview of quality of service for distributed objects. In *Proceedings of the Fifth IEEE conference on Dual Use*, May 1995.
23. J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1), 1997.

Resource Control for Java Database Extensions

Grzegorz Czajkowski, Tobias Mayr, Praveen Seshadri, Thorsten von Eicken

Department of Computer Science, Cornell University
{grzes,mayr,praveen,tve}@cs.cornell.edu

Abstract

While object-relational database servers can be extended with user-defined functions (UDFs), the security of the server may be compromised by these extensions. The use of Java to implement the UDFs is promising because it addresses some security concerns. However, it still permits interference between different users through the uncontrolled consumption of resources. In this paper, we explore the use of a Java resource management mechanism (JRes) to monitor resource consumption and enforce usage constraints. JRes enhances the security of the database server in the presence of extensions allowing for (i) detection and neutralization of denial-of-service attacks aimed at resource monopolization, (ii) monitoring resource consumption which enables precise billing of users relying on UDFs, and (iii) obtaining feedback that can be used for adaptive query optimization.

The feedback can be utilized either by the UDFs themselves or by the database system to dynamically modify the query execution plan. Both models have been prototyped in the Cornell Predator database system. We describe the implementation techniques, and present experiments that demonstrate the effects of the adaptive behavior facilitated by JRes. We conclude that, minimally, a database system supporting extensions should have a built-in resource monitoring and controlling mechanism. Moreover, in order to fully exploit information provided by the resource control mechanisms, both the query optimizer and the UDFs themselves should have access to this information.

1. Introduction

There has been much recent interest in using Java to implement database extensions. The SQL-J proposal [SQLJ] describes efforts by database vendors to support user-defined functions (UDFs) written in Java. Java UDFs are considered relevant in environments like internets and intranets, where large numbers of users extend a database server backend. In earlier work [GMS+98], we explored some of the security, portability, and efficiency issues that arise with Java UDFs. The main observation was that although Java UDFs are efficient, they do not solve all the security problems that arise when a server accepts untrusted extensions. Specifically, short of creating a process per UDF, there is no suitable mechanism to prevent one UDF from allocating large amounts of memory or using a large portion of the CPU time. This allows a malicious or buggy UDF to effectively deny service to all the other users of the database system. Another problem directly and negatively affecting deployment of Java-UDF-enabled database systems is the lack of an infrastructure for monitoring resource consumption and billing users for resources consumed by their UDFs.

In this paper, we describe the application of a Java resource accounting interface, JRes [CvE98], to address this issue. JRes has been incorporated into the Cornell Predator database system [Sesh98a] as part of the Jaguar project, and we base our observations on this prototype. To the best of our knowledge the resulting system is the first database where extensibility based on a safe language is augmented with an ability to monitor usage of computational resources (we note that similar concurrent efforts are being made by vendors of several relational systems). In particular, our work further limits the amount of trust that the database server must have with respect to the behavior of extensions. Due to using a safe language, our previous work ensured that the server is protected from extensions and that the extensions are protected from one another. At the same time, the benefits of executing all participating entities in a single address space can be exploited. This paper demonstrates how a class of UDFs that may execute in a database server without affecting the execution of the server or other extensions can be enlarged to contain UDFs with unknown and potentially malicious or unbalanced resource requirements.

Furthermore, we question two implicit assumptions underlying previous work on optimizing queries with user defined functions: (i) that the costs and completion time of invoking a UDF will remain constant over the execution of the entire query, and (ii) that it is possible to provide realistic estimates on the costs of UDFs. A query executing on large tables and using costly UDFs will execute long enough that considerable fluctuations

in resource availability are likely to be observed while the query is running. Consequently, the relative weights associated with different types of resources will change. Expensive UDFs also often execute complex code, making it difficult to accurately predict their cost. Finally, database cost estimates are typically not absolute; rather they simply need to be accurate relative to each other on some cost scale used by the database system developers (and usually not quantified in terms of real time). The user defining a new UDF has no way to position it on this internal cost scale.

Our work addresses some of these concerns. JRes provides feedback for adaptive query optimization by monitoring the use of resources by each UDF. Depending on the adopted system design, either each UDF requests information about resource consumption and adapts its runtime behavior accordingly, or the database server uses the feedback from the resource monitor to adapt the query's execution. Each model is desirable in certain situations, leading to the conclusion that a database system needs to support both models of resource control feedback.

The rest of the paper is structured as follows. An example-based motivation of our work is contained in the next section. This is followed by a description of selected details on Jaguar and JRes - systems used for experimentation in this study. Section 4 outlines a design space of applicability of dynamic resource controlling mechanisms for user defined functions. Section 5 shows how resource-limiting policies can be defined for Java UDFs. Taking advantage of resource availability feedback is discussed in Sections 6 and 7. This is followed by a discussion of related work and finally by conclusions.

2. Motivation

In order to justify the need for management of computational resources in extensible database servers let us consider the following example. An amateur investor is planning future stock acquisitions and has purchased access to a database server that can be extended with used defined functions coded in Java. Among other data, users of the server can access the table `Companies`, which lists firms whose stock is currently sold and bought on the New York Stock Exchange. The table has two columns of interest for the investor: `Name` (the name of a company) and `ClosingPrices`, which is an array of numbers corresponding to company's share prices. The array contains an entry for every day since the company entered the stock market.

The investor wants to find companies that meet all the following requirements: (i) the company is on the market for at least forty days, (ii) the price of a share forty days ago is smaller than the price today, and (iii) on any given day during the last thirty nine days the price has not changed by more than 2% from the previous day. This can be expressed as the following SQL query:

```
SELECT C.Name
FROM Companies C
WHERE LooksPromising(C.ClosingPrices)
```

where `LooksPromising` is a method of an investor-supplied Java class `StockAnalysis`. Such a class can be written by the investor, generated by a tool, or purchased from a software development house. A simple implementation is shown below:

```
public class StockAnalysis {
  private static final int DAYS = 40;
  private static final int VAR = 0.02;

  public static boolean
  LooksPromising(double[] ts) {
    int size = ts.length;
    if (size < DAYS) return false;
    if (ts[size - DAYS] >= ts[size - 1])
      return false;
    for (int i = 1; i < DAYS; i++) {
      double price = ts[size - i + 1];
      double prevPrice = ts[size - i];
      double v =
        (price - prevPrice)/prevPrice;
      if (Math.abs(v) >= VAR)
        return false;
    }
    return true;
  }
}
```

This kind of database extensibility has many benefits. Many complex filters can be coded much easier and more efficiently when using a programming language instead of SQL. UDFs can be used to integrate user-specific algorithms and external data sources. By controlling the use of the network and the file system, and by using protection mechanisms of Java, the server can ensure that its data is not corrupted or compromised. Cryptography-based protocols like Secure Socket Layer [SSL97] can be used to guarantee secure uploading of UDFs to the server. This means that if investors trust the server they can be assured that nobody else will see the code of their UDFs, which can be a concern when substantial effort was expended towards creating them.

However, at the current state of the art of extensible database technologies [GMS+98] several important issues are still not addressed. These problems are discussed in the subsections below. They include dealing with denial-of-service attacks, accounting for resources consumed by a user's particular UDFs, and

supporting system scalability. For extensible databases where the UDFs are executed in the controlled environment of a safe language, these problems, to a large extent, boil down to the ability to monitor computational resources such as main memory, CPU usage, and network resources.

2.1. Denial-of-Service Attacks

The code of `LooksPromising` is not necessarily well behaved. People make mistakes - for instance, a programmer could forget to increment `i` in the `for` loop which can lead to a non-terminating execution of `LooksPromising` for some inputs. In addition to making mistakes, some code is developed with malicious purposes in mind. One could omit incrementing the loop counter on purpose, or, for instance, insert into `LooksPromising` code to allocate an infinite list so that all available main memory is monopolized by a single instance of the UDF. Regardless of whether such programs are created on purpose or unintentionally, they are equally dangerous in that they can monopolize vital resources. Except for a few trivial cases, it is virtually impossible to decide by means of static code analysis if a Java UDF will use more resources than a particular limit. Dynamic mechanisms that constrain resource usage are needed to prevent denial-of-service attacks. Traditional operating systems use hardware protection and coarse-grained process structure to enforce resource limits. Extensible object-relational database environments, in many ways subsuming the role of an operating system, need to provide the same functionality.

2.2. Accounting for Consumed Resources

Some database servers use accounting mechanisms to charge customers for service. The same will likely happen to extensible database servers based on Java. An immediate problem is that no mechanisms exist to enable accounting for resources consumed by Java UDFs. For instance, CPU time and heap memory used by an invocation of `LooksPromising` are unknown, since Java provides no support for gauging their usage.

Ideally, one should be able to run a UDF and obtain a list of all the resources consumed by it. For instance, in the case of `LooksPromising`, the CPU time and maximum amount of memory used during the invocation should be available. This information can be used for profiling the code and for charging investors for resources consumed during the execution of their queries. Obtaining resource consumption traces from a running UDF is valuable for query optimizers.

2.3. Scheduling and Scalability

Another problem with deploying extensible database servers based on safe languages such as Java is the difficulty of managing large numbers of extensions. Since virtually no information about resource consumption can be obtained, the system does not know what UDFs are particularly resource-hungry and which resources will be stressed when a large number of copies of a particular UDFs are executing simultaneously. This potentially leads to unbalanced resource consumption patterns. For instance, let us imagine several thousand CPU-intensive UDFs copies running (or attempting to run) at the same time. If the UDFs do not adapt their behavior, they face the prospect of slow execution, of deadlock, of being stopped temporarily, or even of being killed by the system, depending on the local policy. This is likely to result in wasted resources since queries and/or UDFs will be aborted halfway through. Providing dynamic information about resources available to UDFs allows database systems to implement admission control policies that minimize the number of aborted UDFs. The UDFs themselves may be coded in a smart way to adapt to changing resource demand and supply. However, in order to be able to perform such coding, an infrastructure and an interface that allows the UDFs to learn about the loads during their execution must be provided.

2.4. An Approach to Manage Resources in Extensible Database Servers

The objective of this work is to provide mechanisms for selected components of resource management in an extensible database where UDFs are executed in a single running copy of the Java Virtual Machine. This includes (i) accounting for resource (CPU time, heap memory, network) usage on a per-UDF basis, (ii) setting limits on resources available to particular UDFs, and (iii) providing the ability to define a specific action to be taken when a resource limit is exceeded. To this end we have extended Java and consequently the JVM serving as an extensibility mechanism with a resource accounting interface, called JRes. The extension does not require any changes to the underlying JVM and relies on dynamic bytecode rewriting and a small native component, coded in the C language. As will be demonstrated later in the paper, most of the problems discussed in this section are addressed in our prototype.

3. Selected Details on Jaguar and JRes Environments

This section contains a brief description of features of Jaguar and JRes relevant for the work presented in this

paper. Both systems have been described in detail elsewhere [GMS+98, CvE98].

3.1. Jaguar

The Jaguar project extends the Cornell Predator object-relational database system [Sesh98a] with portable query execution. The goals of the project are two-fold: (a) to migrate client-side query processing into the database server for reasons of efficiency, (b) to migrate server-side query processing to any component of the channel between the server and the ultimate end-user. In short, the project aims to eliminate the artificial server-client boundaries with respect to query execution. The motivation of the project is the next-generation of database applications that will be deployed over the Web. In such applications, a large number of physically distributed end-users working on diverse and mutually independent applications interact with the database server. In this context, portable query execution can translate into greater options for efficient evaluation and consequently reduced user response times.

The Predator database server is written in C++, and permits new extensions (new data types and UDFs, also written in C++). To explore goal (a) of the Jaguar project, the database server has been enhanced with the ability to define UDFs with Java. This provides clients with a portable mechanism with which to specify client-side operations and migrate them to the server. Java seems to be a good choice as a portable language for UDFs, because Java bytecode can be run with security restrictions within the Java Virtual Machine.

In the current implementation, Java functions are invoked from within the server using either Sun's Java Native Interface or Microsoft's Raw Native Interface. The first step is to initialize the Java Virtual Machine (JVM) as a C++ object. Any classes that need to be used are loaded into the JVM using a custom interface. When methods of the classes need to be executed, they are invoked through JNI or RNI, depending which vendor's JVM is currently used. Parameters that need to be passed to Java UDFs must be first mapped to Java objects.

The creation of the JVM is a heavyweight operation. Consequently, a single JVM is created when the database server starts up and is used until server shutdown. Each Java UDF is packaged as a method within its own class. If a query involves a Java UDF, the corresponding class is loaded once for the whole query execution. The translation of data (arguments and results) requires the use of further interfaces of the JVM. Callbacks from the Java UDF to the server occur through the "native method" feature of Java. There are a number of details associated with the implementation of support for Java UDFs. Importantly, security mechanisms can prevent UDFs from performing unauthorized functions.

3.2. JRes

Through JRes, the trusted core of Java-based extensible databases can (i) be informed of all new thread creations, (ii) state an upper limit on memory used by all live objects allocated by a particular thread or thread group, (iii) limit how many bytes of data a thread can send and receive, (iv) limit how much CPU time a thread can consume, and (v) register *overuse callbacks*, that is, actions to be executed whenever any of the limits is exceeded. Both trusted core and untrusted extensions can learn about resource limits and resource usage.

JRes consists of two Java interfaces, one exception, and the class `ResourceManager`.The class defines constants identifying resources and exports several methods. The methods can be divided up into two categories: privileged and general access. The privileged, authenticated methods can be used only by the execution environment (server, browser). Setting and clearing resource limits, setting and invoking overuse callbacks all fall into this category. In the context of this work, this ensures that only the database server itself has privileged access to the resource management subsystem. UDFs are prevented from interfering with the resource management policies of a given system.

The general access JRes methods, available to all entities in the system, allow for querying the resource management subsystem about resource usage of a particular thread or thread group, about resource limits imposed on a thread or a thread group, and about system-wide resource availability.

Overuse callbacks can be coded as arbitrary Java code. Consequently, what they can do is limited by the control mechanisms that are part of the JVM. For instance, it is possible to lower a thread's priority but it is impossible to change the thread-scheduling algorithm. Another limitation is the inability to track memory allocated in the native code. This is due to the fact that most of JRes is implemented through bytecode rewriting.

The design and operation of our current prototype, which combines Jaguar and JRes, is shown in Figure 1. In this example setup, two remote clients submit their queries through a Web interface. The UDF code (i.e. Java classes) is loaded by the Jaguar class loader. The subsequent execution is controlled by what the standard Java Security Manager and the JRes Resource Manager allow.

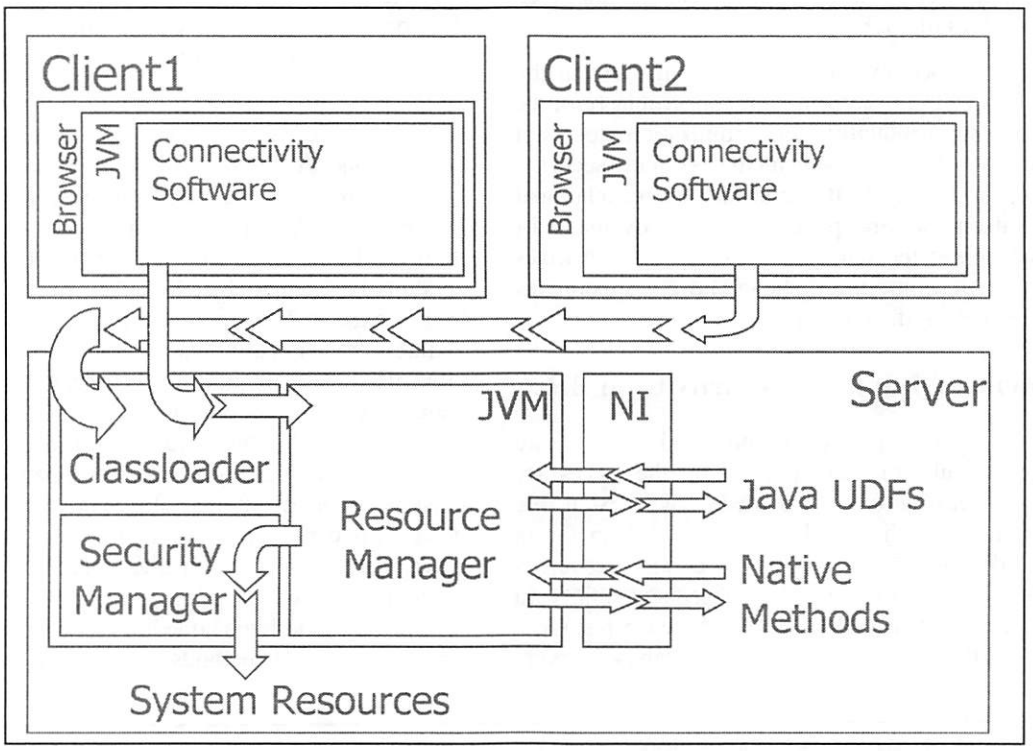


Figure 1. The design and operation of Jaguar extended with Resource Manager.

4. Design Space

Let us take a look at possible dimensions along which a resource monitoring facility can be taken advantage of in an extensible object-relational database system. The first dimension roughly quantifies the UDF programmer's involvement in monitoring the resources. One end of the spectrum is populated by UDFs that monitor their own resource consumption and the resource limits to adjust their execution patterns with respect to changing resource availability. A UDF that dynamically adapts the accuracy of the produced results to the availability of resources forms an example. The other end of the spectrum consists of systems that monitor the resources available to extensions and apply this information to change execution of queries containing UDFs. A database server dynamically reordering conjunctive predicates depending on their resource usage would be placed here.

The other, orthogonal dimension is the domain of application of knowledge about both system-wide and per-UDF resource consumption. One such domain is security - detection of malicious UDFs and preventing denial of service attacks. Another domain is

Functionality	UDFs' involvement	Server's involvement
Optimization	UDFs that query the database server environment in order to adjust their execution and improve performance.	Database servers that optimize query execution through utilizing resource consumption information.
Security	UDFs that query the database server environment in order to avoid using more resources than allowed.	Database servers that monitor resource consumption of UDFs to detect malicious behavior.

Responsibility for using resource information.

Figure 2. Dimensions of applicability of resource monitoring.

optimization, where combining knowledge of resource demands and their availability may lead to improved execution times of UDFs.

A system may occupy more than one quadrant in the outlined space. Using information concerning resource utilization and availability for optimization does not preclude its usage for enhancing system security. Similarly, both the UDFs and an object-relational database itself can independently take advantage of JRes feedback at the same time. Figure 2 summarizes the classification introduced above and gives examples belonging to each of the groups.

5. Enhanced Database Security using JRes

As stated earlier, protection provided by a safe language is only one component of the necessary security infrastructure provided by extensible environments. Another vital part, neglected so far in available designs, is the ability to control resources available to extensions and the subsequent ability to detect and neutralize malicious or otherwise resource-unstable UDFs. Since the class of database servers discussed in this paper falls into the extensible environments category, it is crucial for an unimpeded development and deployment of this data access technology to pay attention to resource monitoring issues.

Figure 3 shows one possible policy that limits each UDF to one thread only. Moreover, such a thread is limited to no more than 50kB of memory and less than 10 milliseconds of CPU time out of every 100 milliseconds. The limits are set whenever a thread creation is detected by JRes. Whenever the memory limit is exceeded, an appropriate exception is thrown. In addition to signalling a problem, this effectively prevents the operation of object creation from completion. Exceeding the time limit results in lowering the offending thread's priority; if the priority cannot be lowered any more, the thread is stopped. It must be pointed out that stopping threads should be dealt with carefully, since threads may own state or other resources, like open files, which may need to be saved or cleaned up appropriately before killing the thread. The underlined code in Figure 3 is a part of JRes (either as defined methods or interface methods to be

```
class ExtensibleDBServerRMP
  implements ThreadRegistrationCallback, OveruseCallback {

  private Object cookie;

  private ExtensibleDBServerRMP(Object cookie) { this.cookie = cookie; }

  public static synchronized void initialize() {
    Object cookie = new Object();
    ResourceManager.initialize(cookie);
    ExtensibleDBServerRMP rmp = new ExtensibleDBServerRMP (cookie);
    ResourceManager.setThreadRegistrationCallback(cookie, rmp);
  }

  public void threadRegistrationNotification(Thread t) {
    if (t.getThreadGroup().getName().equals("system")) { return; }
    if (udfHasThreadsAlready(t)) { stopThread(t); }
    ResourceManager.setLimits(cookie, RESOURCE_CPU, t, 10, 100, this);
    ResourceManager.setLimits(cookie, RESOURCE_MEM, t, 50, 0, this);
  }

  public void resourceUseExceeded(int resType, Thread t, long value) {
    if (resType == RESOURCE_CPU) {
      int priority = t.getPriority();
      if (priority == Thread.MIN_PRIORITY) { stopThread(t); }
      else { t.setPriority(priority - 1); }
    } else if (resType == RESOURCE_MEM) {
      throw new JResResourceExceededException("memory");
    }
  }
}
```

Figure 3. An example resource controlling policy for user defined functions.

defined). The code of several methods is not shown. The details of JRes and its interface have been presented in [CvE98]; the goal of Figure 3 is to demonstrate how resource controlling policies for Java UDFs can be defined.

6. Design of Resource Control Feedback for Java UDFs

The JRes interface allows for retrieving information about current system-wide resource availability and about per-UDF consumption. This information can be used in several ways to improve either overall system performance or the performance of "smart" UDFs. In this section, we describe several scenarios that show usage and applicability of the Jaguar/JRes resource monitoring. In the next section, we demonstrate the performance impact when JRes is used in this fashion.

6.1. Obtaining UDF Costs as a Function of Input Arguments

[Sesh98b] explores optimizations on the boundary between relational query execution and the execution of UDFs and method extensions. The paper identifies four categories of optimization opportunities and studies techniques applicable to each of the categories. An important category requires knowledge of the resource consumption of the UDFs. Our work provides a practical framework in which resource utilization information can be obtained and used for improving query plans. For instance, let us consider the following query

```
SELECT C.Name
FROM Companies C
WHERE
  LooksPromising(C.ClosingPrices) = true
  AND ExternalRating(C.Name) > 0.9
  AND Profitability(C.Name) = "Top"
```

The three UDF predicates are "black boxes" from the viewpoint of both the underlying database and the module managing the extensibility. In order to generate the optimal plan, the query optimizer must know the selectivity and cost of each predicate involved. Thus, an off-line or on-line gathering of performance and selectivity data is necessary in order to provide the query optimizer with the required information. In the example above, some predicates may access the network (for instance, `ExternalRating` may have to communicate with other databases), some may be very CPU-intensive, and others may use large quantities of memory.

Applying JRes off-line to generate a table associating input sizes with execution time, bytes sent and received, and the maximum amount of memory used is simple. However, such a table makes sense only if the input size determines the resource consumption. The process of generating such tables may sometimes uncover that there is simply no correlation between the argument size and the resources consumed by the UDF.

6.2. Dynamic Predicate Reordering Based on Resource Consumption

It is often not possible to execute a query off-line - for instance when it has been submitted by a user during an interactive session with a database server. In such settings, Jaguar augmented with JRes is used to gather dynamic resource profiles. The information can then be fed dynamically to the execution engine, which may change the order of predicate execution based on similar criteria as in the static case.

Dynamic resource monitoring has one advantage over static monitoring - relative values of resources are known, so localized adjustments can be performed better. Let us assume that in the example query from the previous subsection `Profitability` (very CPU-intensive) is applied after the equally selective `ExternalRating` (which consumes large quantities of network bandwidth). The order of predicates will change during the same query execution whenever the system detects that due to the presence of other queries and UDFs in the system there is currently contention for the network while a relatively large amount of CPU time is available. The predicates with high costs, in terms of currently scarce resources, are executed later, benefiting from the selectivity of earlier predicates.

6.3. Dealing with Resource Shortages without Reduced Quality

As described in detail in [Pang94], queries executing in a priority scheduling environment face the prospect of continually having resources taken away and then given back during their lifetime. The same statement is true for UDFs as well, especially for those invoked in queries with long lifetimes; typically, this category would include UDFs operating on large data inputs. Let us take a look at UDFs for which the quality of a result may not suffer but the completion time may worsen. For instance, let us consider a query that invokes a UDF in order to determine whether one image contains another:

```
SELECT P.Name
FROM Paintings P, Cats C
WHERE Contains(P.Image, C.Image) = true
```

The images are stored in a compressed format and `Contains` has to decompress them in order to run a pattern-matching algorithm. If memory is scarce, only parts of images may be decompressed. This will make the pattern-matching operation more time intensive

while the results will be the same, and, more importantly, invocations of `Contains` will not be prematurely aborted because of lack of memory.

6.4. Adjusting Quality of UDF Results when Necessary Resources are Scarce

In some scenarios, adapting to resource scarcity may be accomplished by degrading the quality of output. Examples include faster image operations resulting in worse quality of results that are nevertheless useful for the end user. Another such example can be seen through the eyes of a user of a financial database. Her UDFs return approximations of the standard deviation of an input time series. The CPU time available to any UDF invocation can be limited system-wide in order to make quick response times more likely for a large population of users. In this setting, the UDF must complete without using more resources as given - otherwise, it will be terminated and no result will be produced. Thus, while there is no bound on the length of the time series, the time available to the UDF is bounded. The UDF can query JRes for the CPU time available to itself. This, in turn, can be used to compute the number of entries of the input series that can be processed before using up the quota. If less then the whole series can be processed, it is up to the UDF to decide which ones: the most plausible choices include sampling with a fixed step size or using the most recent section of the time series. The return value may be less precise than whatever could be computed with unlimited resources, but is still a much better alternative than getting nothing back because the UDF's execution has been aborted.

6.5. Exploiting Resource Tradeoffs

In some scenarios, one resource can be traded off for another in order to mask temporary or recurring fluctuation in resource availability. One example has been presented in Section 6.3. Another one is, for instance, a UDF that sends data back directly to the client via a network connection may choose to send compressed results or to send the data "as is". In the first case, more CPU time but less bandwidth is needed; the reverse holds in the second scenario. The most common form of trading resources off for one another is caching, where memory (main or disk) is traded off for whatever resources were consumed to generate cached data. Let us take a look at the following join, where the UDF `Similar` detects a similarity between two time series, retrieved from some other table or from a file system:

```
SELECT D1.name, D2.name
FROM Data D1, Data D2
WHERE Similar(D1.name, D2.name) > 0.7
```

A naïve way of coding `Similar` is to retrieve time series based on the names of arguments, compare inputs, and return the value describing the similarity. However, since the UDF is invoked repeatedly in this query, simple optimizations are possible. If the query is executed using a nested-loop join algorithm (scanning D1, and for each tuple, finding a "matching" tuple in D2), the UDF will be invoked several times with the same first argument. The UDF code could choose to cache the first argument, thereby using memory to reduce CPU and I/O time.

7. Performance Study of Run-Time Adaptation

While previous sections discussed possible uses of resource monitoring mechanisms, this section focuses on quantifying the impact of using JRes in Jaguar. The experimental results presented below were obtained on a Pentium II 300 MHz computer with 128 MB of RAM, running Windows NT Workstation 4.0. The Java Virtual Machine used by Jaguar was Microsoft Visual J++, v. 1.1. Each of the experiments uses a table T with 1000 distinct tuples, each holding two integers, one of which is an identifier for a separately stored time series. The experiments are simple and it can be argued that not very realistic, but they indicate potential performance gains resulting from dynamic monitoring of resource availability and appropriate adaptive behavior

To set the stage for our experiments, let us consider three UDFs: UDF-1, UDF-2 and UDF-3. Each of them takes as an argument an integer identifying a certain time series and returns a boolean value. The costs of these UDFs are considerably larger than the costs of simple predicates (e.g. integer comparisons). The first two UDFs use caching to internally store results of their computations - sorting and computing various statistical moments of time series. UDF-3 does not cache its results and thus its execution time does not depend on the amount of memory available to it. Figure 4 shows the average execution time of each of the UDFs as a function of the amount of memory available per UDF. There were a hundred distinct time series involved; all of them fit into an 800kB cache. Two points are worth stressing here. First, the results of Figure 4 can be easily obtained and can then be used by a static query optimizer. Second, UDF-1 and UDF-2 are examples of user defined functions that utilize a possible resource tradeoff. In this particular case, the tradeoff was increased consumption of main memory (caching) versus reduced need for CPU time.

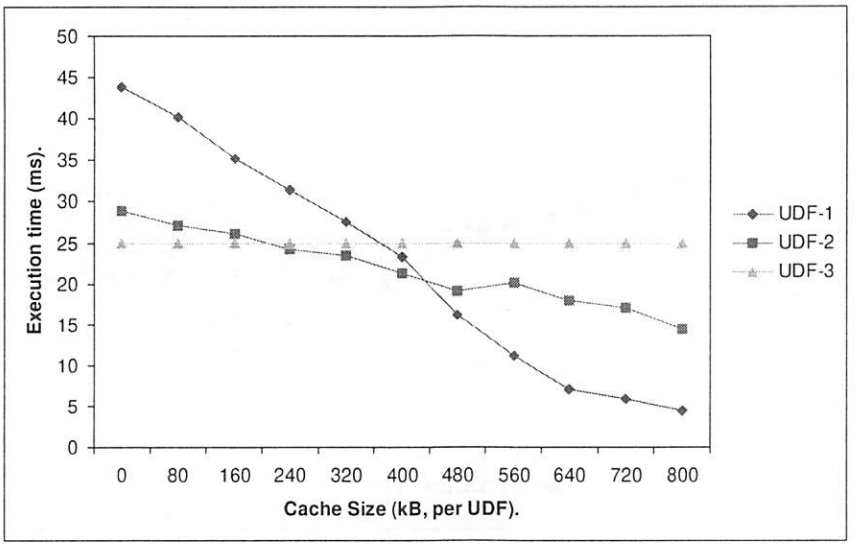


Figure 4. Execution time of three UDFs as a function of available memory.

7.1. Dynamic Predicate Reordering

The three UDFs were coded so that they have the same selectivity (on the average, each of them returns `true` for 30% of its inputs) and in fact always return the same answer if given the same input argument (i.e. whenever UDF1 is true, so are UDF2 and UDF3, and vice versa). Consider the following query:

```
SELECT T.Timeseries
FROM T
WHERE UDF1(T.Timeseries)
  AND UDF2(T.Timeseries)
  AND UDF3(T.Timeseries)
```

The execution time depends on the order in which the predicates are applied and on the amount of memory available to the UDFs . Every nontrivial predicate is associated with a certain cost and a certain selectivity. The latter determines the average ratio of tuples on which the predicate results in true. Selective and cheap predicates should be applied before less selective and more expensive predicates to reduce the overall execution cost. We picked three different evaluation orders: 1-2-3, 2-1-3, and 3-1-2[1], and compared their costs with the cost of a dynamically adapted order. We varied the available cache size, changing the relative costs of the predicates and thus their optimal order. Figure 5 shows the average per-tuple processing time for each of the three given evaluation orders and for an adaptive strategy. The latter monitored available memory and applied this information to dynamically optimize the evaluation order. Incurring a small overhead for the dynamic plan modification, the adaptive strategy always chooses the best order for the predicates.

If the three UDFs were coded as one large UDF invoking the three tests by itself, the reordering could be done inside the UDF. In the case of predicates applied to the same input, it is possible (with a bit of additional work) to re-code them as a single predicate.

7.2. Reordering Join and Selection Operations

Let us now consider the following query, operating on a table T (with 1000 tuples, each of them consisting of two integers; the first one serves as a reference to a stored time series) and a table S (containing 10000 tuples, each of them also consisting of two integers):

```
SELECT *
FROM T, S
WHERE T.a = S.a and UDF1(T.a)
```

Due to the equality predicate used in the join between T and S, the join has a certain selectivity with respect to the table T. The application of UDF-1 can take place either before or after the join, changing the cost of the overall query execution. Applying UDF-1 before the join results in an invocation of UDF-1 on each tuple of T, but reduces the number of tuples of T that have to be joined. On the other hand, applying the join first

[1] Because all three UDFs return the same values on identical arguments, it only matters which predicate is evaluated first: if it returns `false`, the later two are not evaluated; if it returns `true`, all other predicates are evaluated as well. The three picked permutations are equivalent in their complexity to 1-3-2, 2-3-1, and 3-2-1, respectively.

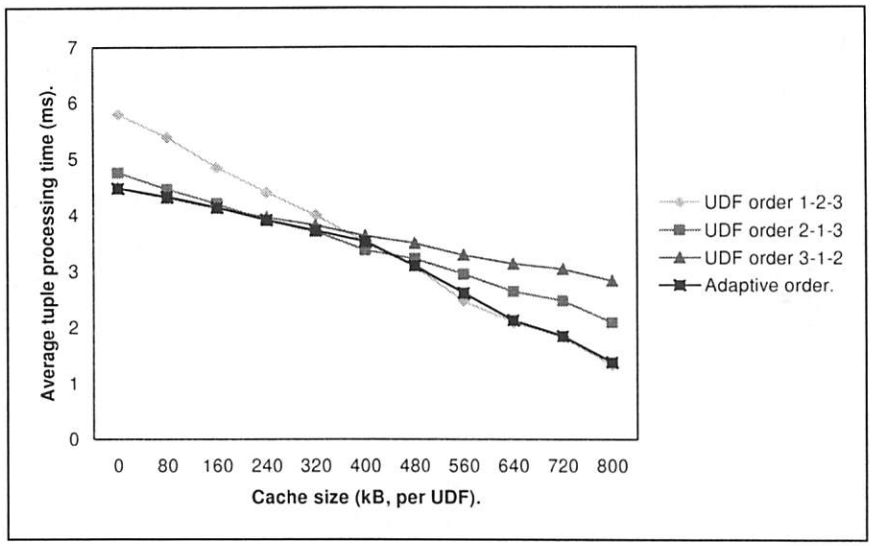


Figure 5. Execution time of different predicate ordering strategies.

requires less invocations of UDF-1, but more tuples are joined. The total cost of the query is different in both cases. Our prototype can change the plan dynamically, during query execution. Figure 6 shows how the two static strategies perform under changing memory availability and contrasts it with the performance of the dynamically adapted plan. The adaptation - applying selection before or after the join - is done similarly to the previous experiment: the resource monitoring information is used by Jaguar to change the plan while it is executing. As Figure 6 demonstrates, the performance gains can be quite substantial when memory availability changes frequently. As in the previous experiment, with a small overhead the adaptive strategy follows the best, hybrid plan. Let us note that in this particular experiment, unlike in the previous one, the query plan reordering can only be the responsibility of the query execution module -- it cannot be taken over by an adaptive UDF.

7.3. Overheads Introduced by JRes

The benefits of on-line resource monitoring come at a price of runtime overheads. For the UDFs used in our experiments, the added execution time overheads are within 3-6%. The overheads are directly proportional to the number of objects allocated by UDFs and in some

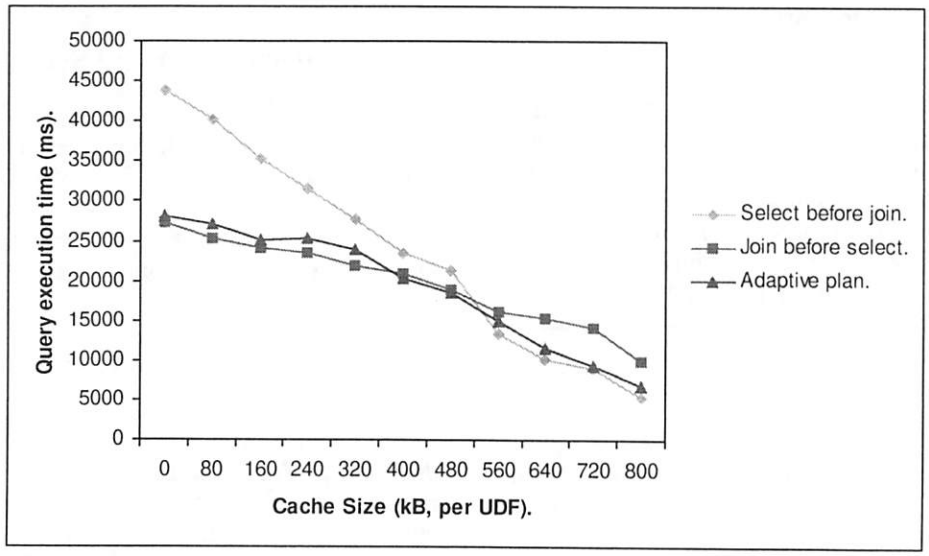


Figure 6. Execution time of different plans.

cases can be substantial [CvE98]. The overheads may be reduced if JRes is integrated into the JVM. Still, increased system security and the ability to adapt both execution plans and UDF execution have to be weighed against the increased execution time.

8. Related Work

Past work related to our research falls into two broad categories: (i) predicting and controlling resource consumption in existing database systems, and (ii) resource accounting and enforcing resource limits in traditional and extensible operating systems and programming languages (a much more detailed discussion of this area can be found in [CvE98]). In this section, we summarize the most important work from these areas influencing our research.

8.1. Database Systems

Several database systems and standards allow the implementation of functions in C, C++ or Java, either as predicates or as general functions. The examples include POSTGRES [SR86], Starburst [HCL+90], Iris [WLH90], and several commercially available systems - for instance Informix, DB2, Oracle 8. The issue of expensive predicate optimization was first raised in the context of POSTGRES [Sto91] and a practically applicable theory addressing the issue was developed in [HS93]. The goal of a recent work of Hellerstein and Naughton [HN97] is to optimize the execution of queries with expensive predicates by caching their arguments and results. The resulting technique, Hybrid Caching, is promising in the presence of repeated invocations of a predicate on the same arguments.

Obtaining realistic estimates of the costs of user defined methods is difficult and quite often imprecise [Hel95]. Typically, it is assumed that, along with estimating selectivity, the creator or user of a UDF will provide a cost estimate as well. Assuming that cost estimates are correct and remain constant throughout the entire execution of the query, it is possible to efficiently generate an optimal plan over the desired execution space [CS96].

Another line of research refines query optimization by focusing on join reordering where an important working assumption is that predicates are zero-cost [IK84, KBZ86, SI92]. A general formulation of query optimization for various buffer sizes can be found in [INS+92]. This runtime parameter is typically unknown before the actual query execution. By constructing various plans in advance, the most appropriate one can be chosen at run-time just before the query is executed, when the available buffer size is known. Another technique helping with estimation of the query size is adaptive sampling [LNS90], where statistical methods are used to predict the result size based on selective runs of the estimated query. Completing joins and sorts under fluctuating availability of main memory has been the subject of recent research by [Pang94].

Dynamic query optimization was incorporated into a commercially available Rdb/VMS system [Ant93]. The research suggests that it is cost-effective to run several local plans simultaneously with proportional speed for a short time, and then select the "best" plan to be run for a long time. An optimization model that assigns the bulk of the optimization effort to compile-time and delays carefully selected optimization decisions until runtime is described in [CG94]. Dynamic plans are constructed at compile-time and the best one is selected at runtime, when cost calculations and comparisons can be performed. The approach guarantees plan optimality. However, none of these approaches deals with unknown and changing costs of user defined functions.

Our work differs from the research mentioned above in our focus on UDFs and on monitoring the environment in which UDFs execute. In addition to providing the ability to run queries off-line to get estimates of their cost, our system constantly monitors resource utilization. This information is available directly both to the UDFs themselves and the query execution module. Both the database system and UDFs can utilize this knowledge directly and dynamically.

8.2. Operating Systems and Programming Languages

Enforcing resource limits has long been a responsibility of operating systems. For instance, many UNIX shells export the `limit` command, which sets resource limitations for the current shell and its child processes. Among others, available CPU time and maximum sizes of data segment, stack segment, and virtual memory can be set. Enforcing resource limits in traditional operating systems is coarse-grained in that the unit of control is an entire process. The enforcement relies on kernel-controlled process scheduling and hardware support for detecting memory overuse.

The architecture of the SPIN extensible operating system allows applications to safely change the operating system's interface and implementation [BSP+95]. SPIN and its extensions are written in Modula-3 and rely on a certifying compiler to guarantee the safety of extensions. The CPU consumption of untrusted extensions can be limited by introducing a time-out. Another example of an extensible operating system concerned with constraining resources consumed by extensions is the VINO kernel [SES+96]. VINO uses software fault isolation as its safety mechanism and a lightweight transaction system to cope with resource hoarding. Timeouts are associated

with time-constrained resources. If an extension holds such a resource for too long, it is terminated. The transactional support is used to restore the system to a consistent state after aborting an extension.

Except for the ability to manipulate thread priorities and invoke garbage collection, Java programmers are not given any interface to control resource usage of programs. Several extensions to Java attempt to alleviate this problem, but none of them share the goals of JRes. For instance, the Java Web Server [JWS97] provides an administrator interface that displays resource usage in a coarse-grained manner, e.g. the number of running threads; however, the information about memory or CPU used by each individual thread is not accessible. PERC (a real-time implementation of Java) [Nils96] provides an API for obtaining guaranteed execution time and assuring resource availability. While the goal of real-time systems is to ensure that applications obtain *at least* as many resources as necessary, the goal of JRes is to ensure that programs do not exceed their resource limits.

9. Conclusions

The security and functionality of an extensible database server can be enhanced by providing resource-controlling mechanisms in the language used for creating user-defined functions. Because of the combination of portability, security, and object-orientation, Java emerges as a premier language for creating extensible environments. Our work evaluates, in the context of extensible database servers, a resource controlling interface we have developed for general purpose Java programs. To the best of our knowledge, no database system supporting UDFs (or, for that matter, no other extensible server system that does not rely on hardware protection) currently provides the functionality we have added to Jaguar. The presented description of the system design, the evaluation of resource monitoring, and the provision of mechanisms for adaptive behavior are important steps towards practical extensible servers.

In particular, our work further limits the amount of trust that the database server must have with respect to the behavior of extensions. The standard JVM controls access of UDFs to security-sensitive resources such as files and network. This paper demonstrates that a class of UDFs that may execute in a database server without affecting the execution of the server or other extensions has been enlarged to contain UDFs with unknown and potentially malicious or unbalanced resource requirements. Moreover, the paper shows that the execution cost of a UDF may depend on the dynamic supply of computational resources. Thus, changing query plan dynamically, during the query execution, is necessary to achieve optimal performance. Jaguar extended with JRes provides appropriate mechanisms for achieving this goal.

Even though this work is carried out in the context of an extensible object-relational database and Java extensions, the conclusions generalize to any system where Java code dynamically extends an execution environment, like a Web browser or an extensible Web server. The security can be enhanced and performance concerns can be addressed in such environments in a similar way to our prototype implementation analyzed in the paper.

10. Acknowledgements

This research is funded by DARPA ITO Contract ONR-N00014-92-J-1866, NSF Contract CDA-9024600, and ASC-8902827, Thorsten von Eicken's NSF Career Grant CCR-9702755, a Sloan Foundation Fellowship and Intel hardware donations. In addition, the work on the Cornell Jaguar project was funded in part through an IBM Faculty Development award and a Microsoft research grant to Praveen Seshadri, through a contract with Rome Air Force Labs (F30602-98-C-0266) and through a grant from the National Science Foundation (IIS-9812020).

11. References

[Ant93] G. Antoshenkov. *Dynamic Query Optimization in Rdb/VMS.* Intl. Conference on Data Engineering, 1993.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM SOSP, Copper Mountain, CO, Dec. 1995.

[CG94] R. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. ACM SIGMOD '94.

[CS96] S. Chauduri, K. Shim. *Optimization of Queries with User-defined Predicates.* 23rd International Conference on Very Large Database Systems, 1996.

[CvE98] G. Czajkowski, and T. von Eicken. *JRes: A Resource Accounting Interface for Java.* ACM OOPSLA'98, Vancouver, BC, October 1998.

[GMS+98] M. Godfrey, T. Mayr, P. Seshadri, T. von Eicken. *Secure and Portable Database Extensibility.* ACM SIGMOD'98.

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. *Starburst Mid-Flight: As the Dust Clears.* IEEE Trans. on Knowledge and Data Engineering, March 1993.

[Hel95] J. Hellerstein. *Optimization and Execution Techniques for Queries with Expensive Methods.* Ph.D. Thesis, University of Wisconsin-Madison, May 1995.

[HS93] J. Hellerstein, and M. Stonebraker. *Predicate Migration: Optimizing Queries with Expensive Predicates.* ACM SIGMOD'93.

[HN97] J. Hellerstein, and J. Naughton. *Query Execution Techniques for Caching Expensive Methods*. ACM SIGMOD '97.

[IK84] T. Ibaraki, T. Kameda. *Optimal Nesting for Computing N-Relational Joins*. ACM Transactions on Database Systems, October 1984.

[INS+92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. *Parametric Query Optimization*. Proc. 18th International Conference on Very Large Database Systems, 1992.

[JWS97] JavaSoft. *Java Web Server*. http://jserv.javasoft.com.

[KBZ86] R. Krishnamurthy, H. Boral and C. Zaniolo. *Optimization of Nonrecursive Queries*. 12th International Conference on Very Large Database Systems, 1986.

[LNS90] R. Lipton, J. Naughton, and D. Schneider. *Practical Selectivity Estimation through Adaptive Sampling*. ACM SIGMOD '90.

[Nils96] K. Nilsen. Issues in the Design and Implementation of Real-Time Java. Java Developer's Journal, 1996.

[Pang94] H. H. Pang. *Query Processing in Firm Real-Time DatabaseSystems*. Ph.D. Thesis, University of Wisconsin Madison, March 1994.

[SES+96] M. Seltzer, Y. Endo, C. Small, and K. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*. 2nd USENIX OSDI, Seattle, WA, October, 1996.

[Sesh98a] P. Seshadri. *Enhanced Data Types in PREDATOR*. VLDB Journal 1998.

[Sesh98b] P. Seshadri. *Relational Query Optimization with Enhanced ADTs*. Technical Report TR98-1693, Cornell University, Computer Science Department, Ithaca, NY, 1998.

[SI92] A. Swami and B. Iyer. *A Polynomial Time Algorithm for Optimizing Join Queries*. Research Report RJ8812, IBM Almaden Research Center, June 1992.

[SSL97] Netscape. *Secure Socket Layer*. http://developer.netscape.com/docs.

[SQLJ] Oracle. *SQL: Embedded SQL for Java - Tutorial*. http://www.oracle.com/st/products/jdbc/sqlj/sql_specs.html.

[Sto91] M. Stonebraker. Managing Persistent Objects in a Multi-Level Store. ACM SIGMOD '91.

[SR86] M. Stonebraker, and L. Rowe. *The Design of POSTGRES*. ACM SIGMOD'86.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. *The Iris Architecture and Implementation*. IEEE Transactions on Knowledge and Data Engineering, March 1990.

Address Translation Strategies in the Texas Persistent Store

Sheetal V. Kakkad*
Somerset Design Center
Motorola
Austin, Texas, USA
svkakkad@acm.org

Paul R. Wilson†
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas, USA
wilson@cs.utexas.edu

Abstract

Texas is a highly portable, high-performance persistent object store that can be used with conventional compilers and operating systems, without the need for a preprocessor or special operating system privileges. Texas uses *pointer swizzling at page fault time* as its primary address translation mechanism, translating addresses from a persistent format into conventional virtual addresses for an entire page at a time as it is loaded into memory.

Existing classifications of persistent systems typically focus only on address translation taxonomies based on semantics that we consider to be confusing and ambiguous. Instead, we contend that the *granularity choices* for design issues are much more important because they facilitate classification of different systems in an unambiguous manner unlike the taxonomies based only on address translation. We have identified five primary design issues that we believe are relevant in this context. We describe these design issues in detail and present a new general classification for persistence based on the granularity choices for these issues.

Although the coarse granularity of pointer swizzling at page fault time is efficient in most case, it is sometimes desirable to use finer-grained techniques. We examine different issues related to fine-grained address translation mechanisms, and discuss why these are not suitable as general-purpose address translation techniques. Instead, we argue for a mixed-granularity approach where a coarse-grained mechanism is used as the primary address translation scheme, and a fine-grained approach is used for specialized data structures that are less suitable for the coarse-grained approach.

We have incorporated fine-grained address translation in Texas using the C++ *smart pointer* idiom, allowing programmers to choose the kind of pointer used for any data member in a particular class definition. This approach maintains the important features of the system: persistence that is orthogonal to type, high performance with standard compilers and operating systems, suitability for huge shared address spaces across heterogeneous platforms, and the ability to optimize away pointer swizzling costs when the persistent store is smaller than the hardware-supported virtual address size.

1 Introduction

The Texas Persistent Store provides portable, high-performance persistence for C++ [16, 8], using pointer swizzling at page fault time [23, 8] to translate addresses from persistent format into virtual memory addresses. Texas is designed to implement and promote *orthogonal persistence* [1, 2]. Orthogonal persistent systems require that any arbitrary object can be made persistent without regard to its type; that is, persistence is viewed as the storage class[1] of an object rather than as a property of its type. In other words, persistence is a property of individual objects, not of their classes or types, and any object can be made persistent regardless of its type. In contrast, *class-based* persistent systems require that any type or class that may be instantiated to create persistent objects *must* inherit from a top-level abstract "persistence" class, which defines the *interface* for saving and restoring data from a persistent object store.

Texas uses *pointer swizzling at page fault time* as the primary address translation technique. When a page is brought into memory, all pointers in the page are identified and translated (or swizzled) into raw virtual ad-

*The work reported in this paper was performed as part of the author's doctoral research at The University of Texas at Austin.

†This research was supported by grants from the IBM Corporation and the National Science Foundation.

[1] A storage class describes how an object is stored. For example, the storage class of an automatic variable in C or C++ corresponds to the stack because the object is typically allocated on the data stack, and its lifetime is bounded by the scope in which it was allocated.

dresses. If the corresponding referents are not already in memory, virtual address space is *reserved* for them (using normal virtual memory protections), allowing for the address translation to be completed successfully. As the application dereferences pointers into non-resident pages, these are intercepted (using virtual memory access protection violations) and the data is loaded from the persistent store, causing further pointer swizzling and (potential) address space reservation for references to other non-resident data. Since running programs only see pointers in their normal hardware-supported format, conventionally-compiled code can execute at full speed without any special pointer format checks.

This page-wise address translation scheme has several advantages. One is that it exploits spatial locality of reference, allowing a single virtual memory protection violation to trigger the translation of all persistent addresses in a page. Another is that off-the-shelf compilers can be used, exploiting virtual memory protections and trap handling features available to normal user processes under most modern operating systems.

However, as with any other scheme that exploits locality of reference, it is possible for some programs to exhibit access patterns that are unfavorable to a coarse-grained scheme; for example, sparse access to large indexing structures may unnecessarily reserve address space with page-wise address translation than with more conventional pointer-at-a-time strategies. It is desirable to get the best of both worlds by combining coarse-grained and fine-grained address translation in a single system.

In Texas, we currently support a fine-grained address translation strategy by using *smart pointers* [17, 7, 12] that can replace normal pointers where necessary. Such pointers are ignored by the usual swizzling mechanism when a page is loaded into memory; instead, each pointer is individually translated as it is dereferenced using overloaded operator implementations. The mixed-granularity approach works well, as shown by experimental results gathered using the OO1 benchmark [4, 5].

The remainder of this paper is structured as follows. In Section 2, we describe existing well-known address translation taxonomies put forth by other researchers, and motivate the need for a general classification of persistence presented in Section 3. In Section 4, we discuss issues about fine-grained address translation techniques, and why we believe that a pure fine-grained approach is not suitable for general use. We describe the implementation of mixed-granularity address translation in Texas in Section 5 and the corresponding performance results in Section 6, before wrapping up in Section 7.

2 Address Translation Taxonomies

Persistence has been an active research area for over a decade and several taxonomies for pointer swizzling techniques have been proposed [13, 9, 11, 19]. In this section, we describe important details about each of these taxonomies and highlight various similarities and differences among them. We also use this as a basis to provide motivation for a general classification of persistent systems based on granularity issues, which we describe in Section 3.

2.1 Eager vs. Lazy Swizzling

Moss [13] describes one of the first studies of different address translation approaches, and the associated terminology developed for classifying these techniques. The primary classification is in terms of "eager" and "lazy" swizzling based on *when* the address translation is performed. Typically, eager swizzling schemes swizzle an entire collection of objects together, where the size of the collection is somehow bounded. That is, the need to check pointer formats, and the associated overhead, is avoided by performing aggressive swizzling. In contrast, lazy swizzling schemes follow an incremental approach by using dynamic checks for unswizzled objects. There is no predetermined or bounded collection of objects that must be swizzled together. Instead, the execution dynamically locates and swizzles new objects depending on the access patterns of applications.

Other researchers [9, 11] have also used classifications along similar lines in their own studies. However, we consider this classification to be ambiguous and confusing for general use. It does not clearly identify the fundamental issue—the *granularity* of address translation—that is important in this context. For example, consider pointer swizzling at page fault time using this classification. By definition, we swizzle all pointers in a virtual memory page as it is loaded into memory and an application is never allowed to "see" any untranslated pointers. There is no need to explicitly check the format of a pointer before using it, making pointer swizzling at page fault time an eager swizzling scheme. On the other hand, the basic approach is incremental in nature; swizzling is performed one page at a time and only on demand, making it a lazy swizzling scheme as per the original definition.

In general, a scheme that is "lazy" at one granularity is likely to be "eager" at another granularity. For example,

a page-wise swizzling mechanism is lazy at the granularity of pages because it only swizzles one page at a time, but eager at the granularity of objects because it swizzles multiple objects—an entire page's worth—at one time. As such, we contend that the granularity at which address translation is performed is the fundamental issue.

2.2 Node-Marking vs. Edge-Marking Schemes

Moss also describes another classification based on the strategy used for distinguishing between resident and non-resident data in the incremental approach. The persistent heap and various data structures are viewed as a directed graph, where data objects represent *nodes* and pointers between objects represent *edges* that connect the nodes. The address translation mechanisms are then classified as either *node-marking* or *edge-marking* schemes.

Figure 1 shows the basic structure for node-marking and edge-marking schemes. As the name suggests, *edge-marking* schemes mark the graph edges—the pointers between objects—to indicate whether they have been translated into local format and reference resident objects. In contrast, *node-marking* schemes guarantee that all references in resident objects are always translated, and the graph nodes themselves are marked to indicate whether they are non-resident. In other words, edges are guaranteed to be valid local references but the actual referents may be non-resident. Note that the marking applies only to non-resident entities, that is, either to nodes that are non-resident or to (untranslated) edges that reference non-resident nodes.

Figure 2 shows a classic implementation of a node-marking scheme; non-resident nodes are "marked" as such by using *proxy* objects, that is, pseudo-objects that stand in for non-resident persistent objects and contain their corresponding persistent identifiers. When an object is loaded from the database, all references contained in that object must be swizzled as per the definition of node-marking—pointers to resident objects are swizzled normally while pointers to non-resident objects are swizzled into references to proxy objects. When the application follows a reference to a proxy object, the system loads the referent (F in the figure) from the database and updates the proxy object to reference the newly-resident object (Figure 2b). Alternatively, the proxy object may be bypassed by overwriting the (old) reference to it with a pointer to the newly-resident object; if there are no other references to it, the proxy object may (eventually) be reclaimed by the system. Note, however, that the compiled code must still check for the presence of proxy objects on *every* pointer dereference because of the possibility that any pointer *may* reference a proxy object. This adds continual checking overhead, even when all pointers directly reference data objects without intervening proxy objects.

Pointer swizzling at page fault time is essentially a node-marking scheme, because swizzled pointers *always* correspond to valid virtual memory addresses, while the referents are distinguished on the basis of residency. However, it differs in an important way from the normal approach—unlike the classic implementation, there are no *explicit* proxy objects for non-resident in pointer swizzling at page fault time. Instead, access-protected virtual address space pages *act* as proxy objects.[2] As the application progresses and more data is loaded into memory, the pages that were previously protected are now unprotected because they contain valid data. The major advantage of this approach is that there is no need to reclaim proxy objects (because none exist); consequently, there are no further indirections that must be dealt with by compiled code, avoiding continual format checks that would otherwise be necessary.

2.3 General Classification for Persistence

We have seen that existing classifications focus only on address translation techniques. While address translation is an important issue, it constitutes only one of several design issues that must be considered when implementing persistence. We have identified a set of design issues that we believe are fundamental to efficient implementation of any persistence mechanism. We believe that a specific combination of these issues can be used to characterize any particular implementation. In effect, we are proposing a general classification scheme based on *granularities of fundamental design aspects*.

A classification based on "eager" and "lazy" swizzling is ambiguous, because it does not attack the problem at the right level of abstraction. The real issue in the distinction between lazy and eager swizzling is the size of the unit of storage for which address translation is performed. This can range from as small as a single reference (as in Moss's "pure lazy swizzling" approach) to a virtual memory page (as in pointer swizzling at page fault time), or even as large as an entire database (as in Moss's "pure eager swizzling" approach).[3]

[2] In fact, unmapped virtual address space pages can also serve the same purpose.

[3] While crude, this is actually not uncommon. Traditionally, Lisp

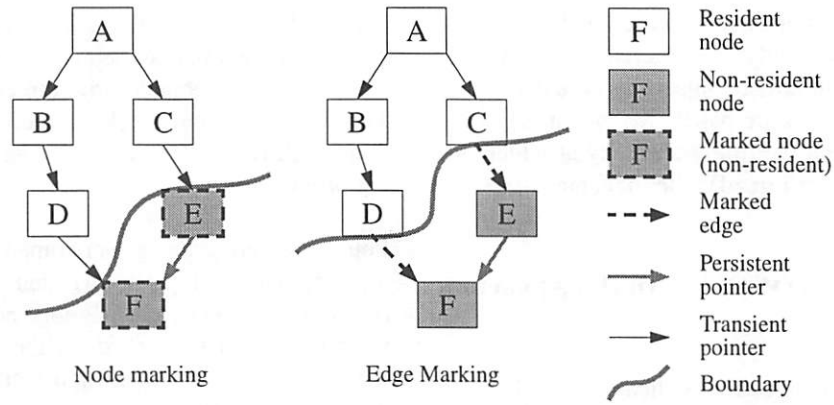


Figure 1: Node-marking and edge-marking schemes

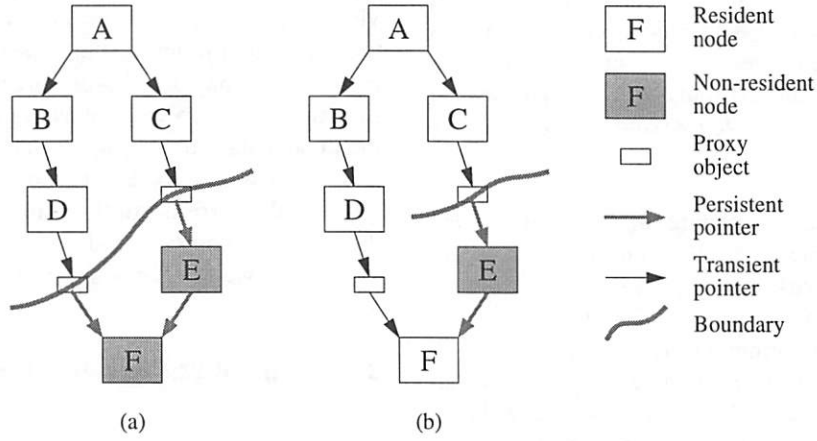


Figure 2: Node-marking scheme using proxy objects

We believe that it is preferable to consider address translation (and other design issues) from the perspective of a *granularity choice* rather than an *ad hoc* classification based on confusing translation semantics. In fact, the ambiguity arises primarily because the classifications either do not clearly identify the granularity, or, because they unnecessarily adhere to a single predetermined granularity. Discussing all design issues in terms of granularity choices provides a uniform framework for identifying the consequence of each design issue on the performance and flexibility of the resulting persistence mechanism. This is preferable to ambiguous classifications such as eager and lazy swizzling because many schemes are both "eager" and "lazy" at different scales, along several dimensions.

and Smalltalk systems have supported the saving and restoring of entire heap images in a "big inhale" relocation.

3 Granularity Choices for Persistence

We have identified a set of five design issues (including address translation) that are relevant to the implementation of a persistence mechanism. Each of these issues can be resolved by making a specific granularity choice that is independent of the choice for any other issue. The combination of granularity choices for the different issues can then be used to characterize persistent systems. The specific design issues that we describe in this section are the granularities of *address translation*, *address mapping*, *data fetching*, *data caching* and *checkpointing*. In the remainder of this section, we define and discuss each issue in detail[4] and also present the rationale behind the granularity choices for these issues in our implementation of orthogonal persistence in Texas.

[4]Note that while we describe each issue individually, these granularity choices are strongly related. It is possible (and quite likely) that a system may make the same granularity choice on multiple issues for various reasons.

To a first approximation, the basic unit for all granularity choices in Texas is a virtual memory page, because pointer swizzling at page fault time relies heavily on virtual memory facilities, especially to trigger data transfer and address translation. The choice of a virtual memory page as the basic granularity unit allows us to exploit conventional virtual memories, and avoid expensive runtime software checks in compiled code, taking advantage of user-level memory protection facilities of most modern operating systems. Sometimes, however, it is necessary to change the granularity choice for a particular issue to accommodate the special needs of unusual situations. It is possible to address these issues at a different granularity in a way that integrates gracefully into the general framework of Texas.

3.1 Address Translation

The granularity of *address translation* is the smallest unit of storage within which all pointers are translated from persistent (long) format to virtual memory (short) format. In general, the spectrum of possible values can range from a single pointer to an entire page or more.

The granularity of address translation in Texas is typically a virtual memory page, for coarse-grained translation implemented via pointer swizzling at page fault time. The use of virtual memory pages has several advantages in terms of overall efficiency because we use virtual memory hardware to check residency of the referents. In addition, we also rely on the application's spatial locality of reference to amortize the costs of protection faults and swizzling entire pages.

As described in Section 5, it is possible to implement a fine-grained address translation mechanism for special situations where the coarse-grained approaches are unsuitable, because of poor locality of reference in the application. Since Texas allows fine-grained translation on individual pointers, the granularity of address translation in those cases would be a single pointer.

3.2 Address Mapping

A related choice is the granularity of *address mapping*, which is defined as the smallest unit of addressed data (from the persistent store) that can be mapped independently to an area of the virtual address space.

To a first approximation, this is a virtual memory page in Texas because any page of persistent data can be mapped into any arbitrary page of the virtual address space of a process. A major benefit of page-wise mapping is the savings in table sizes; we only need to maintain tables that contain mappings from persistent to virtual addresses and vice versa on a page-wise basis, rather than (much larger) tables for recording the locations of individual objects. This reduces both the space and time costs of maintaining the address translation information.

However, the granularity of address mapping is bigger than a page in the case of large (multi-page) objects. When a pointer to (or into) a large object is swizzled, virtual address space must be reserved for all pages that the large object overlaps. This reservation of multiple pages is necessary to ensure that normal indexing and pointer arithmetic works as expected within objects that cross page boundaries. The granularity of address mapping is then equivalent to the number of pages occupied by the large object.

3.3 Data Fetching

As the name suggests, the granularity of *data fetching* is the smallest unit of storage that is loaded from the persistent store into virtual memory. As with the two granularities presented above, we use a virtual memory page for this purpose in the current implementation of Texas. The primary motivation for making this choice was simplicity and ease of implementation, and the fact that this correlated well with the default granularity choices for other design issues in our implementation.

It is possible to change the granularity of fetching without affecting any other granularity choices. In essence, we can implement our own prefetching to preload data from the persistent store. This may actually be desirable for some applications when using raw unbuffered I/O instead of normal file I/O [8]. Depending on the access characteristics of the application and the dataset size, the overall I/O costs can be reduced by prefetching several (consecutive) pages instead of a single faulted-on page. In general, the granularity of data fetching is intimately tied to the I/O strategy that is selected in the implementation.

3.4 Data Caching

The granularity of *data caching* is defined as the smallest unit of storage that is cached in virtual memory. For Texas, the granularity of caching is a single virtual mem-

ory page, because Texas relies exclusively on the virtual memory system for caching persistent data.

A persistent page is usually cached in a *virtual memory* page as far as Texas is concerned. The virtual memory system determines whether the page actually resides in RAM (i.e., physical memory) or on disk (i.e., swap space) without any intervention from Texas. This is quite different from some other persistent storage systems which directly manage physical memory and control the mapping of persistent data into main memory. In general, Texas moves data between a persistent store and the virtual memory *without regard to the distinction between virtual pages in RAM and on disk*; that is, virtual memory caching is left up to the underlying virtual memory system, which does its job in the normal way.

It is, of course, possible to change this behavior such that Texas directly manages physical memory. However, we believe that this is unnecessary, and may even be undesirable, for most applications. The fact that Texas behaves like any normal application with respect to virtual memory replacement may be advantageous for most purposes because it prevents any particular application from monopolizing system resources (RAM in this case). As such, applications using Texas are just normal programs, requiring no special privileges or resources; they "play well with others" rather than locking up large amounts of RAM as many database and persistent systems do.

3.5 Checkpointing

Finally, we consider the granularity of *checkpointing*, which is defined as the smallest unit of storage that is written to non-volatile media for the purpose of saving recovery information to protect against failures and crashes.

Texas uses virtual memory protections to detect pages that are modified by the application between checkpoints. Therefore, the default unit of checkpointing in the usual case is a virtual memory page. Texas employs a simple write-ahead logging scheme to support checkpointing and recovery—at checkpoint time, modified pages are written to a log on stable storage before the actual database is updated [16].

The granularity of checkpointing can be refined by the use of sub-page logging. The approach relies on a page "diffing" technique that we originally proposed in [16]. The basic idea is to save clean versions of pages before they are modified by the application; the original (clean) and modified (dirty) versions of a page can then be compared to detect the exact sub-page areas that are actually updated by the application and only those "diffs" are logged to stable storage. This technique can be used to reduce the amount of I/O at checkpoint time, subject to the application's locality characteristics. The granularity of checkpointing in this case is equivalent to the size of the "diffs" which are saved to stable storage.[5]

Another enhancement to the checkpointing mechanism is to maintain the log in a compressed format. As the checkpoint-related data is streamed to disk, we can intervene to perform some inline compression using specialized algorithms tuned to heap data. Further research has been initiated in this area [24] and initial results indicate that the I/O cost can be reduced by about a factor of two, and that data can be compressed fast enough to double the effective disk bandwidth on current machines. As CPU speeds continue to increase fast than disk speeds, the cost of compression shrinks exponentially relative to cost of disk I/O. Further reduction in costs is also possible with improved compression algorithms and adaptive techniques.

4 Fine-grained Address Translation

There are several factors that motivated us to develop a coarse-grained mechanism over a fine-grained approach when implementing pointer swizzling at page fault time in Texas. The primary motivation is the fact that we wanted to exploit existing hardware to avoid expensive residency checks in software. However, we believe that there are also other factors against using a fine-grained approach as the primary address translation mechanism. In this section, we discuss fine-grained address translation techniques and why we believe that they are not practical for high-performance implementations in terms of efficiency and complexity.

Overall, fine-grained address translation techniques are likely to incur various hidden costs that have not been measured and quantified in previous research. In general, we have found most current fine-grained schemes appear to be slower than pointer swizzling at page fault time in terms of the basic address translation performance.

[5] The basic "diffing" technique has been implemented in the context of QuickStore [19]; preliminary results are encouraging, although more investigation is required.

4.1 Basic Costs

Fine-grained address translation techniques usually incur some inherent costs due to their basic implementation strategy. These costs can be divided into the usual time and space components, as well as less tangible components related to implementation complexity. We believe that these costs are likely to be on the order of tens of percent, even in well-engineered systems with custom compilers and fine-tuned run-time systems. Some of the typical costs incurred in a fine-grained approach are as follows:

- A major component of the total cost can be attributed to *pointer validity* checks. These checks can include both *swizzling* checks and *residency* checks. A swizzling check is used to verify whether a reference is translated into valid local format or not[6] while a residency check verifies whether the referent is resident and accessible. These two checks, while conceptually independent of each other, are typically combined in implementations of fine-grained schemes.
- Another important component of the overall cost is related to the implementation of a custom object replacement policy, which is typically required because physical memory is directly managed by the persistence mechanism. This cost is usually directly proportional to the rate of execution because it requires a read barrier.[7] We discuss this further in the next subsection.
- As resident objects are evicted from memory, a proportional cost is usually incurred in invalidating references to the evicted objects. This is necessary for maintaining *referential integrity* by avoiding "dangling pointers." This cost is directly proportional to the rate of eviction and locality characteristics of the application.
- By definition, fine-grained translation techniques permit references to be in different formats during application execution. This requires that pointers be checked to ensure that they are in the right format before they can be used, even for simple equality checks. It may also be necessary to check transient pointers, depending on the underlying implementation strategy. As such, there is a continual pointer format checking cost that is also dependent on the rate of execution and pointer use.
- Finally, it is possible to incur other costs that exist mainly because of unusually constrained object and/or pointer representations used by the system. For example, accessing an object through an indirection via a proxy object is likely to require additional instructions.[8] Another example is the increased complexity required for handling languages features such as interior pointers.[9]

Note that all cost factors described above do not necessarily contribute to the overall performance penalty in every fine-grained address translation mechanism. However, the basic costs are usually present in some form in most systems.

4.2 Object Replacement

Fine-grained address translation schemes typically require that the persistence mechanism directly manage physical memory because persistent data are usually loaded into memory on a per-object basis.[10] Therefore, it is usually necessary to implement a custom object replacement policy as part of the persistence mechanism. This affects not only the overall cost but also the implementation complexity.

A read barrier is typically implemented for every object that resides in memory. The usual action for a read barrier is to set one bit per object for maintaining recency information about object references to aid the object replacement policy. The read barrier may be implemented in software by preceding each object read with a call to the routine that sets the special bit for that object. Compiled code then contains extra instructions—usually inserted by the compiler—to implement the read barrier. The read barrier is typically expensive on stock hardware because, in the usual case, *all* read requests must be intercepted and recorded. It is known that one in about ten instructions is a *pointer store* (i.e., a write into a pointer) in Lisp systems that support compilation. Since read actions are more common than write actions, we estimate

[6] For example, all swizzled pointers in Texas *must* contain valid virtual memory address values.

[7] The term *read barrier*, borrowed from garbage collection research [21], is used to denote a trigger that is activated on every read operation. A corresponding term, *write barrier*, is used to denote triggers that are activated for every write operation.

[8] Some systems use crude replacement and/or checkpointing policies to simplify integration with persistence and garbage collection mechanisms. These may incur additional costs due to the choice of suboptimal policies.

[9] *Interior pointers* are those that point inside the bodies of objects rather than at their heads.

[10] The data are usually read from the persistent store into a buffer (granularity of data fetching) in terms of pages for minimizing I/O overhead. However, only the objects required are copied from the buffer into memory (granularity of data caching).

that between 5 and 20 percent of total instructions in an application usually correspond to a read from a pointer. The exact number obviously varies by application, and more importantly, by the source language; for example, it is likely to be higher in heap-oriented languages such as Java. It may be possible to use data flow analysis during compilation such that the read barrier can be optimized away for some object references; such analysis is, however, hard to implement.

The object replacement policy also interferes with general swizzling, especially if an edge-marking technique is being used. In such cases, the object cannot be evicted from memory without first invalidating all edges that reference it. This obviously requires knowledge about references to the object being evicted. Kemper and Kossman [9] solve this by using a per-object data structure known as a *Reverse Reference List (RRL)* to maintain a set of back-pointers to all objects that reference a given object. McAuliffe and Solomon [11] use a different data structure, called the *swizzle table*, a fixed-size hash table that maintains a list of all swizzled pointers in the system. Both these approaches are generally unfavorable because they increase the storage requirements (essentially doubling the number of pointers at the minimum) and the implementation complexity.

4.3 Discussion

One of the problems in evaluating different fine-grained translation mechanisms is the lack of good measurements of system costs and other related costs in these implementations. The few measurements that do exist correspond to interpreted systems (except the E system [14, 15]) and usually underestimate the costs for a high-performance language implementation. For example, a 30% overhead in a slow (interpreted) implementation may be acceptable for that system, but will certainly be unacceptable as a 300% overhead when the execution speed is improved up by a factor of ten using a state-of-the-art compiler.

Another cost factor for fine-grained techniques that has generally been overlooked is the cost of maintaining mapping tables for translating between the persistent and transient pointer formats. Since fine-grained schemes typically translate one pointer at a time, the mapping tables must contain one entry per pointer. This is likely to significantly increase the size of the mapping table, making it harder to manipulate efficiently.

We believe that the E system [14, 15] is probably the fastest fine-grained scheme that is comparable to a coarse-grained address translation scheme; however, it still falls short in terms of performance. Based on the results presented in [19], E is about 48% slower than transient C/C++ for hot traversals of the OO1 database benchmark [4, 5].[11] This is a fairly significant considering that the overhead of our system is *zero* for hot traversals and much smaller (less than 5%) otherwise [8].

We believe that there are several reasons why it is likely to be quite difficult to drastically reduce the overheads of fine-grained techniques. Some of these are:

- Several of the basic costs cannot be changed or reduced easily. For example, the pointer validity and format checks, which are an integral part of fine-grained address translation, cannot be optimized away.
- There is a general performance penalty (maintaining and searching large hash tables, etc.) that is typically independent of the checking cost itself. As mapping tables get larger, it will be more expensive to probe and update them, especially because locality effects enter the overall picture.[12]
- Complex data-flow analysis and code generation techniques are required to optimize some of the costs associated with the read barrier used in the implementation. Furthermore, such extra optimizations may cause unwanted code bloat.
- Although the residency property can be treated as a type so that Self-style optimizations [6] can be applied to eliminate residency checking, it is not easy to do so; unlike types, residency may change across procedure calls depending on the dynamic run-time state of the application. As such, residency check elimination is fundamentally a non-local problem that depends on complex analysis of control flow and data flow.

Based on these arguments, we believe that fine-grained translation techniques are comparatively not as attractive for high-performance implementations of persistence mechanisms.

Taking the other side of the argument, however, it can certainly be said that fine-grained mechanisms have their

[11] The hot traversals are ideal for this purpose because they represent operations on data that have already been faulted into memory, thereby avoiding performance impacts related to differences in loading patterns, etc.

[12] Hash tables are known to have extremely poor locality because, by their very nature, they "scatter" related data in different buckets.

advantages. A primary one is the potential savings in I/O because fine-grained schemes can fetch data only as necessary. There are at least two other benefits over coarse-grained approaches:

- fine-grained schemes can support reclustering of objects within pages, and
- the checks required for fine-grained address translation may also be able to support other fine-grained features (such as locking, transactions, etc.) at little extra cost.

In principle, fine-grained schemes can recluster data over short intervals of time compared to coarse-grained schemes. However, clustering algorithms are themselves an interesting topic for research, and further studies are necessary for conclusive proof. We also make another observation that fine-grained techniques are attractive for unusually-sophisticated systems, e.g., those supporting fine-grained concurrent transactions. Inevitably, this will incur an appreciable run-time cost, even if that cost is "billed" to multiple desirable features. Such costs may be reduced in the future if fine-grained checking is supported in hardware.

5 Mixed-granularity Address Translation in Texas

Pointer swizzling at page fault time usually provides good performance for most applications with good locality of reference. However, applications that exhibit poor locality of reference, especially those with large sparsely-accessed index data structures, may not produce best results with such coarse-grained translation mechanisms. Applications that access big multi-way index trees are a good example; usually, such applications sparsely access the index tree, that is, only a few paths are followed down from the root. If the tree nodes are large and have a high fanout, the first access to a node will cause all those pointers to be swizzled, and possibly reserve several pages of virtual address space. However, most of this swizzling is probably unnecessary since only a few pointers will be dereferenced.

The solution is to provide a fine-grained address translation mechanism which translates pointers individually, instead of doing it a page at a time. Unlike the coarse-grained mechanism where the swizzling was triggered by an access-protection violation, the actual translation of a pointer may be triggered by one of two events—either when it is "found"[13] or when it is dereferenced.

There are many ways of implementing a fine-grained (pointer-wise) address translation mechanism as we described above. We have selected an implementation strategy that remains consistent with our goals of portability and compatibility with existing off-the-shelf compilers, by using the C++ *smart pointer* abstraction [17, 7, 12]. Below, we first briefly explain this abstraction and then describe how we use it for implementing fine-grained translation in Texas. We also discuss how both fine-grained and coarse-grained schemes can coexist to create a mixed-granularity environment.

5.1 Smart Pointers

A smart pointer is a special C++ parameterized class such that instances of this class behave like regular pointers. Smart pointers support all standard pointer operations such as dereference, cast, indexing etc. However, since they are implemented using a C++ class with overloaded operators supporting these pointer operations, it is possible to execute arbitrary code as part of any such operation. While smart pointers were originally used in garbage collectors to implement write barriers [22, 21], they are also suitable for implementing address translation; the overloaded pointer dereference operations (via the "*" and "->" operators) can implement the necessary translation from persistent pointers into transient pointers.

A smart pointer class declaration is typically of the following form:

```
template <class T> class Ptr
{
 public:
  Ptr (T *p = NULL); // constructor
  ~Ptr ();           // destructor
  T& operator * ();  // dereference
  T *operator -> (); // dereference
  operator T * ();   // cast to 'T *'
  ...
};
```

Given the above declaration of a smart pointer class, we can then use it as follows:

[13] A pointer is "found" when its location becomes known. This is similar to the notion of "swizzling upon discovery" as described in [20].

```
class Node;          // assume defined
Node *node_p;        // regular pointer
Ptr<Node> node_sp; // smart pointer
...
node_p->some_method();
node_sp->some_method();
```

Note that we have only shown some of the operators in the declaration. Also, we avoid describing the private data members of the smart pointer because the interface is much more important than the internal representation; it does not matter *how* the class is structured as long as the interface is implemented correctly. In fact, as will be clear from our discussion about variations in fine-grained address translation mechanisms, the smart pointer will need to be implemented differently for different situations and implementation choices.

Smart pointers were designed with the goal of transparently replacing regular pointers (except for declarations), and providing additional flexibility because arbitrary code can be executed for every pointer operation. In essence, it is an attempt to introduce limited (compile-time) reflection [10] into C++ for builtin data types (i.e., pointers).[14] However, as described in [7], it is impossible to truly replace the functionality of regular pointers in a completely transparent fashion. Part of the problem stems from some of the inconsistencies in the language definition and unspecified implementation dependence. Thus, we do not advocate smart pointers for arbitrary usage across the board, but they are useful in situations where further control is required over pointer operations.

5.2 Fine-grained Address Translation

In order to implement fine-grained address translation in Texas, we must swizzle individual pointers, instead of entire pages at a time, thereby reducing the consumption of virtual address space for sparsely-accessed data structures with high fanout. By using smart pointers for this purpose, we allow the programmer to easily choose data structures that are swizzled on a per-pointer basis, without requiring any inherent changes in the implementation of the basic swizzling mechanism.

Note that although the pointers are swizzled individually, the granularity of data fetching is still a page, not individual objects, to avoid excessive I/O costs. Below we describe at least two possible ways to handle fine-grained address translation, and discuss why we choose one over the other.

5.2.1 Fine-grained Swizzling

A straightforward way of implementing fine-grained address translation is to cache the translated address value in the pointer field itself; we call this *fine-grained swizzling*, because the pointer value is cached after being translated.[15] We chose not to follow this approach because of a few problems with the basic technique.

First, fine-grained swizzling incurs checking overhead for every pointer dereference; the first dereference will check and swizzle the pointer, while future dereferences will check (and find) that the swizzled virtual address is already available and can be used directly. A more significant problem is presented by equality checks (*a la* the C++ == operator)—when two smart pointers are compared, the comparison can only be made after ensuring that both pointers are in the same representation, that is, either both are persistent addresses or both are virtual addresses. In the worst-case scenario, the pointers will be in different representations, and one of them will have to be swizzled before the check can complete. Thus, a simple equality check, on average, can become more expensive than desired.

One solution is to make the pointer field large enough to store both persistent and virtual address values, as in E [14, 15]. In the current context, the smart pointer internal representation could be extended such that it can hold both the pointer fields. This technique avoids the overhead on equality checks, which can be implemented by simply comparing persistent addresses without regard to swizzling, at the expense of additional storage.

Unfortunately, a more serious problem with fine-grained swizzling is presented by its peculiar interaction with checkpointing. When a persistent pointer is swizzled, the virtual address has to be cached in the pointer field (either E-style or otherwise), that is, we must *modify* the pointer. Since virtual memory protections are used to detect updates initiated by the application for checkpointing purposes, updating a smart pointer to cache the swizzled address will generate "false positives" for updates, causing unnecessary checkpointing. We could work around this problem by first resetting the permissions on the page, swizzling (and caching) the pointer,

[14] C++ already provides limited reflective capabilities in the form of operator overloading for user-defined types and classes. However, this fails to support completely transparent redefinition of pointer operations in arbitrary situations.

[15] The term "swizzling" implies that the translated address is cached, as opposed to discarded after use.

and then restoring the permissions on the page. However, this is very slow on average because it requires kernel intervention to change page protections.

5.2.2 Translations at Each Use

We have seen that a simple fine-grained swizzling mechanism is not as desirable because of its unusual interactions with the operating system and the virtual memory system. However, we can slightly modify the basic technique and overcome most of the disadvantages without losing any of the benefits.

The solution is to implement smart pointers that are translated on *every* use and avoid any caching of the translated value. In other words, these smart pointers hold only the persistent addresses, and must be translated every time they are dereferenced because the virtual addresses are not cached. Equality checks do not incur any overhead because the pointer fields are always in the same representation and can be compared directly.

Pointer dereferences also do not incur any additional checking overhead. The cost of translating at each use does not add much overhead to the overall cost, and is usually amortized over other "work" done by the application; that is, the application may dereference a smart pointer and then do some computation with the resulting target object before dereferencing another smart pointer.

The advantage of this approach is that the pointer fields do not need to be modified because the translated address values are never cached, and all unwanted interactions with checkpointing and the virtual memory system are avoided. Of course, this approach is still unsuitable as a general swizzling mechanism compared to the pointer swizzling at page fault time for reasons described in Section 4.

5.3 Combining Coarse-grained and Fine-grained Address Translation

It is possible to implement a mixed-granularity address translation scheme that consists of both coarse-grained pointer swizzling and fine-grained address translation. The interaction of swizzling with data structures such as B-trees can be handled through the use of the smart pointer abstraction. The details of a fine-grained address translation scheme are hidden, making the approach partially reflective.

We have implemented mixed-granularity address translation in Texas by combining a fine-grained approach using smart pointers that are translated at every use, along with the standard coarse-grained approach. This allows better programmer control over the choice of data structures for which fine-grained address translation is used, while maintaining the overall performance of pointer swizzling at page fault time.

6 Performance Measurements

We present our experimental results for different address translation granularities using the standard OO1 database benchmark [4, 5] with some minor variations as the workload for our experimental measurements. We first briefly explain the rationale for choosing the OO1 benchmark for our performance measurements, then describe the experimental design followed by the actual results, and finally end with a summary.

6.1 Benchmark Choice

Most performance measurements and analysis of persistent object systems (and object-oriented database systems) have been done using *synthetic benchmarks* instead of using real applications. There are two reasons for this: first, there are few large, realistic applications that exercise all persistence mechanisms of the underlying system and of those that exist, few are available for general use; and second, it is typically extremely hard to adapt a large piece of code to any given persistence mechanism without having a detailed understanding of the application.

The OO1 and OO7 [3] benchmarks have become quite popular among various benchmarks, and have been used widely for measuring the performance of persistent systems. However, we posit that these benchmarks are *not representative* of typical real-world applications, because they have not been validated against applications in the domain they represent; other researchers [18] have also reached similar conclusions. As such, the experimental results from these benchmarks should be interpreted with caution. The apparently "empirical" nature of these "experimental" results is likely to lull people into relying on the results more than appropriate. It is important to always remember that while the results are obtained empirically, they are ultimately derived from a synthetic benchmark and are only as good as the map-

ping of benchmark behavior onto real applications.

Although OO1 is a crude benchmark and does not strongly correspond to a real application, we use it for several reasons. First, OO1 is simple for measuring raw performance of pointer traversals (which is what we are interested in) and is fairly amenable to modifications for different address translation granularities. Use of a synthetic benchmark is also appropriate in this situation because our performance is very good in some cases (i.e., zero overhead when there is no faulting) and dependent on the rate of faulting (usually minimal overhead compared to I/O costs) for other cases. As such, crude benchmarking is the most practical way to measure performance of different components of our system because it is easy to separate our costs from those of the underlying benchmark; this is usually more difficult with a real application. Further discussion on sythetic benchmarks and their applicability is available in [8].

6.2 Experimental Design

The benchmark database is made up of a set of *part objects* interconnected to each other. The benchmark specifies two database sizes based on the number of parts stored in the database—a *small database* containing 20,000 parts and a *large database* containing 200,000 parts—to allow performance measurements of a system when the entire database is small enough to fit into main memory and compare it with situations where the database is larger than the available memory.

The parts are indexed by unique *part numbers* associated with each part.[16] Each part is "connected" via a direct link to exactly three other parts, chosen partially randomly to produce some locality of reference. In particular, 90% of the connections are to "nearby" 1% of parts where "nearness" is defined in terms of part numbers, that is, a given part is considered to be "near" other parts if those parts have part numbers that are numerically close to the number of this part. The remaining 10% of the connections are to (uniformly) randomly-chosen parts.

We use the OO1 benchmark traversal operation (perform a depth-first traversal of all connected parts starting from a randomly-chosen part and traversing up to seven levels deep for a total of 3280 parts including possible duplicates, and invoke an empty procedure on each visited part) for our performance measurements. Each *traversal set* contains a total of 45 traversals split as follows: the first traversal is the *cold* traversal (when *no data* is cached in memory), the next 34 are *warm* traversals (as *more and more data* is cached in memory) and finally the last 10 are *hot* traversals (when *all data* is cached in memory).[17] We use a random number generator to ensure that each warm traversal selects a new "root" part as the initial starting point, thus visiting a mostly-different set of parts in each traversal.

[16]The benchmark specification does not define a data structure that must be used for the index; we used a B+ tree for all our experiments.

6.3 Experimental Results

We present results for the OO1 traversal operations corresponding to different address translation granularities for the data structures used during the traversals. In particular, we are interested in three different address translation granularities, namely *coarse-grained*, *mixed-granularity* and *fine-grained* strategies. The following table describes the types of pointers used for each granularity and the corresponding key in the results.

Granularity	Type(s) of pointers	Key
coarse	all language-supported	all-raw
mixed	smart for index	smart-index
fine	all smart	all-smart

We use CPU time[18] instead of absolute real time because the difference in performance is primarily due to differences in faulting and swizzling, and allocating address space for reserved pages. Unfortunately, CPU-time timers on most operating systems have a coarse granularity (typically in several milliseconds), and it would be impossible to measure any reasonable differences in the performance due to a change in the address translation granularity because our overheads are very small. Thus, we use an older SPARCstation ELC, which is slow enough to offset the coarse granularity of the timers, while providing reasonable results.

Figure 3 presents the CPU time for all traversals in an entire traversal set run on a large database. As expected, the cost for "all-raw" case (coarse-grained address translation) is the highest for the first 15 or so traversals. This is not unusual because the coarse-grained address translation scheme swizzles all pointers in the faulted-on pages and reserves many pages that may never be

[17]This is different from the standard benchmark specification containing only 20 traversals (split as 1 cold, 9 warm, and 10 hot traversals); we run more warm traversals because we believe that 9 traversals are not sufficient to provide meaningful results, especially for the large database case.

[18]We refer to the sum of *user* and *system* time as the *CPU time*.

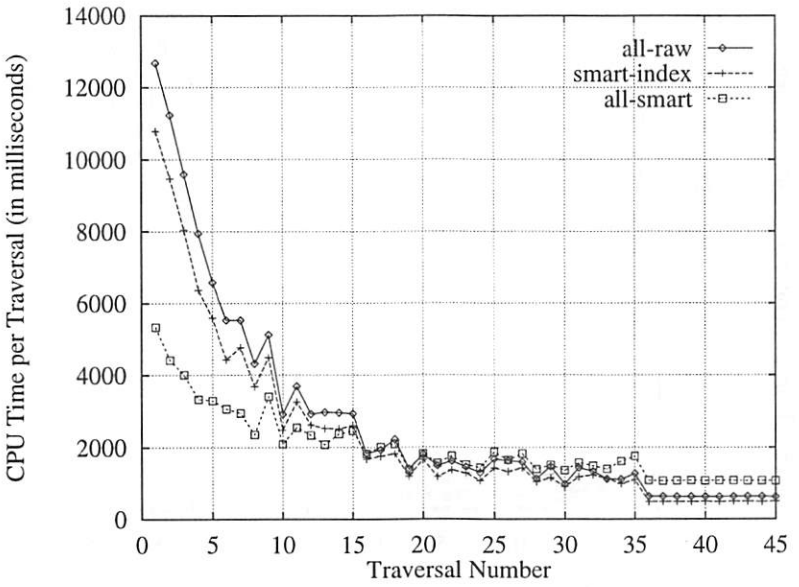


Figure 3: CPU time for traversal on large database

used by the application. This is exacerbated by the poor locality of reference in the benchmark traversals as many pages of the database are accessed during the initial traversals, causing a large number of pages to be reserved. The number of new pages swizzled decreases as the cache warms up, and we see the corresponding reduction in the CPU time.

Note that the cost for the "all-smart" case (fine-grained address translation) is the lowest for the first 15 traversals. Again, this is expected because the address translation scheme does not swizzle any pointers in a page when it is faulted in because they are all smart pointers that must be translated at every use. Finally, the CPU time for the "smart-index" case (mixed-granularity address translation) falls between the other two cases for the first 15 traversals. This is also reasonable because only the index structure contains smart pointers, and each traversal uses this index only once (to select the root part for the traversal). This cost is only slightly less than the "all-raw" case because our B+ tree implementation generated a tree that was only three levels deep, reducing the number of smart pointers that had to be translated for each traversal.

Now consider the hot traversals (36 through 45). The first thing to note is that the CPU time for the "all-smart" case is higher than that for the other two cases. This is because smart pointers impose a continual overhead for each pointer dereference, and this cost is incurred even if the target object is resident. In contrast, the "all-raw" case has zero overhead for hot traversals.[19]

Figure 4 shows the corresponding results for the small database, where only the first 3 or 4 traversals contain faulting and swizzling.[20] Once again, a phenomenon similar to the one in large database results can be seen in the current results, but only for the initial traversals. In particular, the CPU time is highest for the "all-raw" case and lowest for the "all-smart" case. Also as before, the two granularities swap their positions for the hot traversals; the "all-smart" case is more expensive because of the continual translation overhead at every use. Finally, as expected, the "all-raw" and "smart-index" results are identical for hot traversals because no index pointers are dereferenced.

6.4 Summary

The results presented above support our assertion that fine-grained address translation can be effectively used for data structures with high fanout that are less conducive for a coarse-grained scheme. At the same time, a pure fine-grained approach is not the best performing as the primary address translation mechanism in Texas because of various overheads associated with it.

One problem with using the OO1 benchmark is that the operations do not perform any real computation (unlike

[19]The "smart-index" results should be identical to the "all-raw" results for hot traversals because there is no index lookup, and no smart pointers need to be translated. We attribute the difference between the hot results in these two cases to caching effects.

[20]Most of the database is memory-resident within the first few traversals because of the extremely poor locality characteristics in the connections.

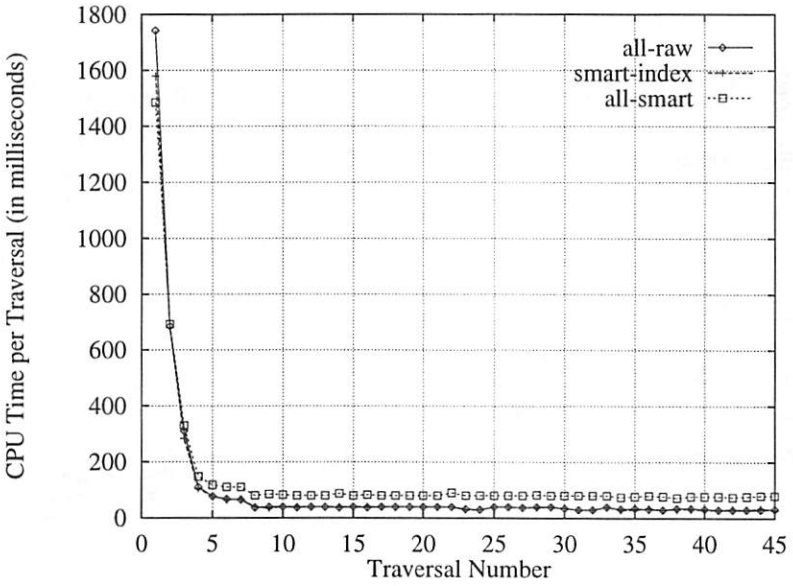


Figure 4: CPU time for traversal on small database

in actual applications) with objects that are traversed. As such, the cost of fine-grained translation is highlighted as a larger component of the total cost than it typically would be in an actual application that performs real "work" on data objects as they are traversed.

7 Conclusions

We presented a discussion on address translation strategies, both in the context of the Texas persistent store and for general persistence implementations. We also proposed a new classification for persistence in terms of granularity choices for fundamental design issues rather than using taxonomies based only on address translation semantics, and discussed each choice that we made in Texas.

We also discussed issues related to fine-grained address translation, including their inherent costs that make them unsuitable as the primary address translation mechanism in a persistence implementation. Instead, we discussed how a mixed-granularity approach can be used to selectively incorporate fine-grained address translation in the application.

We presented our implementation of mixed-granularity address translation in Texas which combines the C++ smart pointer idiom for the fine-grained translation component with the normal pointer swizzling at page fault time mechanism for the coarse-grained translation component, while maintaining portability and compatibility of the system.

Our basic performance results using the OO1 benchmark have shown that the mixed-granularity approach works well for applications with data structures that do not provide the best performance with a pure coarse-grained approach. However, further performance measurements are necessary, especially using real applications instead of synthetic benchmarks which do not always model reality very well.

References

[1] Malcolm P. Atkinson, Peter J. Bailey, Ken J. Chisholm, W. Paul Cockshott, and Ron Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, December 1983.

[2] Malcolm P. Atkinson and Ron Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington DC., June 1993. ACM Press.

[4] R. G. G. Cattell. An Engineering Database Benchmark. In Jim Gray, editor, *The Benchmark Hand-*

book for Database and Transaction Processing Systems. Morgan Kaufmann, 1991.

[5] Rick G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[6] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language*. PhD thesis, Stanford University, March 1992.

[7] Daniel Ross Edelson. Smart Pointers: They're Smart, But They're Not Pointers. In *USENIX C++ Conference*, pages 1–19, Portland, Oregon, August 1992. USENIX Association.

[8] Sheetal V. Kakkad. *Address Translation and Storage Management for Persistent Object Stores*. PhD thesis, The University of Texas at Austin, Austin. Texas, December 1997. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/kakkad-dissertation.ps.gz`.

[9] Alfons Kemper and Donald Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal*, 4(3):519–566, July 1995.

[10] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[11] Mark L. McAuliffe and Marvin H. Solomon. A Trace-Based Simulation of Pointer Swizzling Techniques. In *Proceedings of the International Conference on Database Engineering*, pages 52–61, Taipei, Taiwan, March 1995. IEEE.

[12] Scott Meyers. Smart Pointers. *C++ Report*, 1996. Article series published from April through December. Available at `http://www.aristeia.com/magazines.html`.

[13] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[14] Joel E. Richardson and Michael J. Carey. Persistence in the E Language: Issues and Implementation. *Software Practice and Experience*, 19(12):1115–1150, December 1989.

[15] Daniel T. Schuh, Michael J. Carey, and David J. DeWitt. Persistence in E Revisited—Implementation Experiences. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, September 1990.

[16] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag.

[17] Bjarne Stroustrup. The Evolution of C++, 1985 to 1987. In *USENIX C++ Workshop*, pages 1–22. USENIX Association, 1987.

[18] Ashutosh Tiwary, Vivek R. Narasayya, and Henry M. Levy. Evaluation of OO7 as a System and an Application Benchmark. In *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas, October 1995.

[19] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin, Madison, Wisonsin, 1994.

[20] Seth J. White and David J. Dewitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, October 1992. Morgan Kaufmann.

[21] Paul R. Wilson. Garbage Collection. *ACM Computing Surveys*. Expanded version of [22]. Available at `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. In revision.

[22] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[23] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[24] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, California, April 1999. USENIX Association.

JMAS: A Java-Based Mobile Actor System for Distributed Parallel Computation

Legand L. Burge III *
Systems and Computer Science
Howard University
Washington, DC 20059
blegand@scs.howard.edu

K. M. George
Computer Science Department
Oklahoma State University
Stillwater, Oklahoma 74078
kmg@a.cs.okstate.edu

Abstract

JMAS is a prototype network computing infrastructure based on mobile actors [10] using Java technology. JMAS requires a programming style different from commonly used approaches to distributed computing. JMAS allows a programmer to create mobile actors, initialize their behaviors, and send them messages using constructs provided by the JMAS Mobile Actor API. Applications are decomposed by the programmer into small, self-contained sub-computations and distributed among a virtual network of Distributed Run-Time Managers (*D-RTM*); which execute and manage all mobile computations. This system is well suited for course grain computations for network computing clusters. Performance evaluation is done using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem.

Keywords: Distributed systems, parallel computing, actor model, mobile agents, actors, network computing

1 Introduction

Multicomputers represent the most promising developments in computer architecture due to their economic cost and scalability. With the creation of faster digital high bandwidth integrated networks, heterogeneous multicomputers are becoming an appealing vehicle for parallel computing, redefining the concept of supercomputing. As these high bandwidth connections become available, they shrink distances and change our models of computation, storage, and interaction. With the exponential growth of the World Wide Web (*WWW*), the web can be used to exploit global resources, such as CPU cycles, making them available to every user on the Internet [7, 30]. The combined resources of millions of computers on the Internet can be harnessed to form a powerful global computing infrastructure consisting of workstations, PCs, and supercomputers (Figure 1).

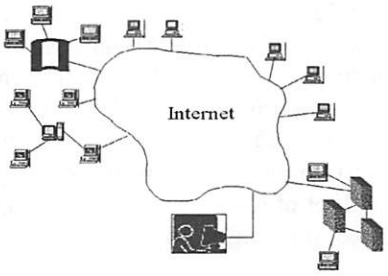


Figure 1 Global Computing Infrastructure.

The vision of integrating network computers into a global computing resource is as old as the Internet[7][23]. Such a system should hide the underlying physical infrastructure from users and from programmers, provide a secure environment for resource owners and users, support access and location of large integrated objects, be fault tolerant, and scale to millions of autonomous hosts. Some recent network computing approaches include CONDOR [29], MPI [24], PVM [31], Piranha [22], NEXUS [28], Network of Workstations (*NOW*) [3], Legion [23], and GLOBUS [20]. These network computing frameworks use low-level communication systems, or high-level dedicated systems. Although these systems offer heterogeneous collaboration of multiple systems in parallel, they involve rather complex maintenance of different binary codes, multiple execution environments, and complex underlying architectures.

Distributed computing over networks, has emerged as a technology with tremendous promise and potential, owing in part to the emergence of the Java Programming Language and the World Wide Web. Recently, researchers have proposed several

*This work was supported in part by the USENIX Student Research Grant

approaches to provide a platform independent Java-based high-performance network computing infrastructure. These include Javalin [16], WebFlow [4], IceT [14], JavaDC [15], Parallel Java [26], Parallel Java Agents [27], ATLAS [5], Charlotte [6], ParaWeb [9], Popcorn [12], and Ninflet [32]. The use of Java as a means for building distributed systems that execute throughout the Internet has also been recently proposed by Chandy et al. [13], Fox et al. [21] and implemented in [33]. Java, because of its platform independence, overcomes the complexity issues of maintaining different binary codes, multiple execution environments, and complex underlying architectures. It offers the basic infrastructure needed to integrate computers connected to the Internet into a distributed computational resource for running parallel applications on numerous anonymous machines.

Mobile agents are a convenient paradigm for distributed computing [8, 18]. The agent specifies when and where to migrate, and the system handles the transmission. This makes mobile agents easier to use than low-level facilities in which the programmer must explicitly handle communication, but more flexible and powerful than schemes such as process migration in which the system decides when to move a program based on a small set of fixed criteria. Mobile agents allow a distributed application to be written as a single program.

In this paper we discuss the design and implementation of a prototype network computing system (*JMAS*) based on the mobile actor model [10] using Java technology [25]. The mobile actor model is a parallel programming paradigm for distributed parallel computing based on mobile agents and the *actor* message passing model [1]. Applications are decomposed by the programmer into small, self-contained subcomputations and distributed among a virtual network of Distributed Run-Time Managers (*D-RTM*); which execute and manage all mobile computations. Lastly, we evaluate the performance of our system, and show that our system is well suited for course grain computations in a network computing environment. Our experiments were ran using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem.

2 The Actor Model

Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing. Actors are characterized by an identity (i.e. mail address), a mailbox, and a current behavior. Moreover, a mail address may be included in messages sent to other actors - this allows those actors to communicate with the actor whose mail address they have received. The ability to communicate mail addresses of actors implies that the interconnection network topology of actors is dynamic. This dynamic interconnection network topology implies that the underlying resources can be represented as actors to build a system architecture. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. Acquaintances represent actors whose mail addresses are known to the actor. Because all actor communication is asynchronous, all messages are buffered in mail queues until the actor is ready to respond to them. Messages sent are guaranteed to be received with an unbounded but finite delay.

Each actor may be thought of as having two aspects that characterize their behavior:

1. its **acquaintances** which is the finite collection of actors that it directly knows about;

2. the **action** it should take when it is sent a message. These actions provide a **primitive** set of operations to:

   - *send* messages asynchronsly to specified actors,
   - *create* actors with specified behaviors, and
   - *become* a new actor, assuming a new behavior to respond to the next message.

The actor primitive operators (i.e. *send, create, and become*) form a simple but powerful set on which to build a wide range of higher-level abstractions and concurrent programming paradigms. Although there is sufficient research supporting the actor model to solve fine/large grain applications on a tightly coupled system, there has been no actor-based solution to solve large scale data intensive distributed applications which may be interconnected by costly communication links. In order to support this environment, locality of reference and resource management (i.e. load balancing) must be addressed; as processes must be able to migrate throughout the system. In the next section, we address the issue of locality of reference and resource management through actor mobility. We present a communication paradigm among mobile agents that incorporates actor-based message passing to support dynamic architecture topologies for distributed parallel computing.

3 The Mobile Actor Paradigm

A *mobile actor* is an actor with the semantics of mobility and navigational autonomy. Navigational autonomy is the degree to which a message can be viewed as an object with its own innate behavior, capable of making decisions about its own destiny. The actor model inherently enforces navigational autonomy allowing addresses of actors to be communicated and thus providing a dynamic interconnection network topology. Such a computing model provides support to deal with non-deterministic problems which require network reconfigurations, non-deterministic communication, and dynamic process coordination. In many practical distributed applications, the over consumption of local resources don't allow computations to be processed efficiently. A more feasible solution would be to migrate the process to least consumed resources, or to move the process to a data server or communication partner in order to reduce network load by accessing a data server or communication partner by local communication. The mobile actor model is a strategy for remote execution and process migration using the actor-message passing paradigm (i.e. for load balancing, and locality of reference of data/behaviors). A remote execution includes the transport and start of execution of a process on a remote location. Process migration includes the transport of process code, execution state, and data of the process; processes may be restarted from their previous state. The execution of computations may migrate across file systems consisting of networks of computers and/or computing clusters.

We extend the actor primitive operations in response to a message with semantics to support actor mobility. The semantics of actor mobility are enforced: upon receipt of a message, or when dynamically creating another actor on a remote location. These extended primitive operations allow computations to migrate after state change.
The behavior of mobile actors consists of two kinds of actions in response to a message:

1. $become_{remote}$ computes a replacement behavior on the local machine and migrates to a location on a remote machine. The migrated actor is characterized by the identity (i.e. it's mail address), and mailbox of a specified location of an actor on a remote machine.

2. $create_{remote}$ a new actor on the local machine and migrate to the remote location, assuming a new behavior to respond to the next message.

4 JMAS: A Java-Based Mobile Actor System

Exploiting the resources of several interconnected computers to form a powerful network computing infrastructure is the goal of this research. Such an infrastructure should provide a single interface to users that provides large amounts of computing power, while hiding from users the fact that the system is composed of hundreds to thousands of machines scattered across the country. Our vision is to create a system in which a user sits at a workstation, and has the illusion of a single very powerful computer. In this section, we discuss the technical issues associated with the construction of a network computing infrastructure which executes mobile actor computations. A mobile actor system is a multi-user, heterogeneous, network computing environment for executing distributed actor-based computations. A mobile actor system must support two basic tasks - the creation and migration of remote actors, and the communication between actors distributed throughout the system. In addition the system should:

- provide language support for the mobile actor programming model,
- provide a single consistent namespace for actors within the system,
- provide an efficient execution schedule between actors maintained on the local machine,
- be able to distribute the load evenly among the machines participating within the distributed system,
- be fault tolerant, and
- be secure.

4.1 JMAS Infrastructure

JMAS is a network computing environment for executing mobile actor computations. JMAS is designed using Java technology [25], and requires a programming style different from commonly used approaches to distributed computing. JMAS allows a programmer to create mobile actors, initialize their behaviors, and send them messages using constructs provided by the JMAS Mobile Actor API. As the computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network. New messages are generated, new actors are created, and existing actors undergo state change. JMAS

also makes mobile actor locality visible to programmers to give them explicit control over actor placement. However, programmers still do not need to keep track of the location to send a message to a mobile actor. Data flow and control flow of a program in JMAS is concurrent and implicit. A programmer thinks in terms of what an actor does, not about how to thread the execution of different actors. Communication of mobile actors is point-to-point, non-blocking, asynchronous, and thus buffered.

4.2 Language Support in JMAS

JMAS is based on the Java Programming Language and Virtual Machine of JDK1.1 [25]. JDK1.1 contains mechanisms that allow objects to be read/written to streams (object serialization), as well as, an API that provides constructs to dynamically build objects at run-time (i.e. Reflection package [*java.lang.reflect*]). We exploit heterogeneity through Java's platform independent (i.e. write once run anywhere) framework. We provide a Mobile Actor API for developing mobile actor applications using the Java Programming Language. Mobile actor programs are compiled using a Java compiler that generates Java bytecode. Java bytecode can be executed on any machine containing a Java Virtual Machine. Actors in JMAS are lightweight processes called threads. The API provides constructs which allow programmers to create mobile actors using static or dynamic placement, to change an actor's state, and send communications to an actor.

4.3 Consistent Mobile Actor Names in JMAS

JMAS implements a simple location-dependent naming strategy tightly coupled with mobile actors within the system. Each mobile actor within the system is given a globally unique identifier. This identifier is bound to only one address by the underlying message system. These bindings may change over time; if for example, a mobile actor migrates to a different machine. In such a case, messages are forwarded to the new location by the underlying message system. It has been shown in [11], that forwarding messages in a distributed system consisting of N machines requires in the worst case $N-1$ message rounds.

4.4 Scheduling and Load Balancing in JMAS

The JVM implements a timeslice schedule of threads on Window95 systems, and a pre-emptive priority-based schedule for UNIX/Windows NT systems. JMAS forces a pre-emptive, priority-based schedule among local threads; regardless of the underlying architecture. The efficiency of an actor-based computation on a loosely coupled architecture depends on where different actors are placed and the communication traffic between them. Thus, the placement and migration of actors can drastically affect the overall performance. We implement a decentralized fault-tolerant load balancing scheme based on the CPU market strategy proposed in [12]. The market strategy is based on CPU-time. Entities within the system consist of *buyers* and *sellers*. A *seller* allows its CPU to be used by other programs. A *buyer* serves as a machine wanting to off-load work to a seller. A meeting place in which buyers and sellers are correlated is known as a *market*. Computations are distributed to seller using a round-robin schedule. This strategy is intended for coarse-grain applications.

4.5 Security in JMAS

Security issues are not addressed in this version of the prototype system. Policies could be enforced to encrypt/decrypt all Java class files and messages sent throughout the system. Use of any strategy will compromise the overall performance of the system.

4.6 Fault Tolerance in JMAS

Machines used within the JMAS infrastructure are fault tolerant to the extent necessary without compromising overall system performance. The limit of our concern is with fail-stop faults of hardware components, and the network. The underlying communication system will guarantee the delivery of messages through the use of reliable, communication-oriented TCP sockets. Further, if a host should fail, then JMAS will remove that host from the current CPU Market configuration.

5 JMAS Architecture

The architecture of JMAS is organized as a series of layers or levels, each one built upon its predecessor (Figure 2). The lowest level(**physical layer**) is the actual physical network, which may consist of a LAN/WAN of PCs and/or workstations. It could also represent a global network such as the Internet. The second layer (**daemon layer**) consists of the collection of daemons residing on all physical machines participating in the distributed system. Each daemon listens

on a reserved communication port receiving communications that could consist of messages or migrating computations. Upon receipt of a communication, it is passed to the third layer. The third layer consists of Distributed Run-Time Managers (**D-RTM**). The D-RTM is responsible for message handling from/to local/remote processes, scheduling and load balancing of processes. The forth layer (**logical layer**), consist of the actual application specific computations on the local machine. Computations are expressed as *mobile actors*. Each actor is encapsulated with a behavior, an identity, a mail queue, and one thread. The logical layer shows each actor and its acquaintances (i.e. A knows about B and C, ...etc).

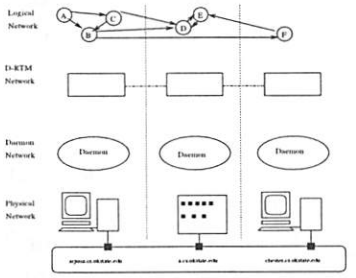


Figure 2. Four Layer Mobile Actor Architecture.

In the following sections, we give a detailed description of the JMAS architecture. In particular, we discuss the components of each layer, and show how Java technology is applied.

5.1 Physical Layer

The physical layer is the actual physical network, which may consist of a LAN/WAN of PCs and/or workstations. These systems are referred to as scalable computer clusters (SCCs), or networks of workstations (NOWs) [3]. Both systems are developed within a trusted environment. Therefore security issues are not a major concern. The disadvantage is that the scalability of these systems is limited to the resources available to the system administrator. The physical layer could also represent interconnected networks of computer clusters.

5.2 Daemon Layer

The daemon layer is implemented as a collection of daemon threads residing on all physical nodes participating in the JMAS distributed environment. The responsibility of the daemon thread is to continuously monitor the network, receiving local/remote communication messages and mobile computations arriving from other machines. JMAS supports a messages-driven model of execution (Figure 3).

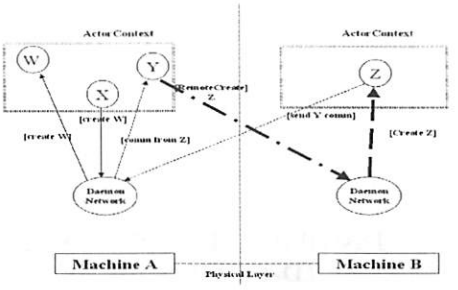


Figure 3. Message-driven model of execution.

There is no local/remote peer-to-peer communication between mobile actors within the system. All communication is routed through a reserved port of a daemon thread residing on the local machine. The reserved port for JMAS is *9000*. Message reception by the daemon thread creates a thread within the actor which executes the specified method with the message as its argument. Only message reception can initiate thread execution. Furthermore, thread execution is atomic. Once successfully launched, a thread executes to completion without blocking.

5.3 Distributed Run-Time Manager

The Distributed Run-Time Manager (D-RTM) is the most complex of the four layers. It is contained within each daemon in the system. Therefore, the daemon layer and D-RTM layer are tightly coupled. The D-RTM contains the basic underlying software that provides the transparent interface to the network computing system. The D-RTM was designed using a layered virtual machine design built on top of the Java Virtual Machine (JVM) using JDK1.1 [25] (Figure 4). The D-RTM has several functions:

- To handle all incoming Tasks (i.e. **Message Handler**)
- To prepare actor processes to run on the local system (i.e. **Actor Context**)
- To load java bytecode (e.g. java objects) from local/remote locations(i.e. **BehvLoader**)
- To schedule local/remote threads using a preemptive, priority schedule (i.e. **Scheduler**),

- To manage the CPU load on the local machine (i.e. **Load Balancer**).

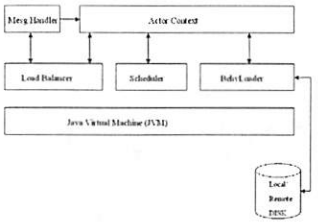


Figure 4. Distributed Run-Time Manager (D-RTM).

5.3.1 Message Handler

The message handler is responsible for routing Tasks which consist of communications to local actors. As illustrated in Figure 5, messages are stored in a table of message queues (i.e. mailboxes). A mailbox could have one or more actors within the local actor context associated to it. We implement the table of mailboxes as a hash table. We use Java's Hashtable class provided by the *java.util* package. Because Java implements its Hashtable as a synchronized object, each access to the Hashtable is atomic. This is very useful for our multi-threaded environment. Each mail address hashes to one mailbox in the table. In order to achieve maximum parallelism, the table is accessed by subprocesses. Messages from a desired mailbox are forwarded asynchronously to actor processes whose identity is denoted by the mail addresses of the mailbox.

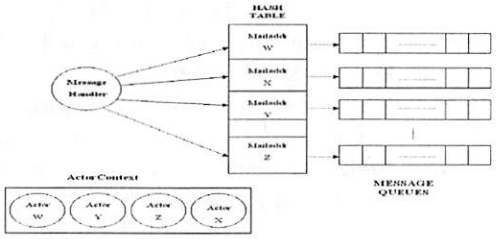


Figure 5. Message Handler.

5.3.2 Actor Context

The Actor Context is responsible for instantiating an object, wrapping the object within a thread, and supplying the thread to the Scheduler. It also maintains a table of system information. Such as:

- The actor Identity
- The current behavior
- The current method (communication being executed)
- The total (idle) time actor waited in ready queue before receiving a communication (msec)
- The total time to load the actor (msec)
- The current running time (msec)

Objects in JMAS are built during runtime. Information about an object during runtime is obtained using Java Reflection [25]. The classes needed to perform these operations are obtained from the *java.lang.reflect* package of the JDK1.1.

5.3.3 Scheduler

JMAS implements a pre-emptive, priority-based scheduler among local threads. Each thread is assigned a priority that can only be changed by the programmer. The thread that has the highest priority is the current running thread. Processes with a lower priority are interrupted. To ensure that starvation does not exists among threads we implement a round-robin schedule among local processes. As illustrated in Figure 6(a), incoming threads or threads instantiated locally, are given a priority–initially low. Threads are then placed into a queue data structure. The scheduler dequeues a thread from the list and assigns it the highest possible priority–causing the this thread to run. After a given time t, the thread is stopped and inserted back into the list. This process continues until all threads within the list terminate (Figure 6(b)). The scheduler could be interrupted by the load balancer; if the CPU reaches its computation threshold. This will cause the current running thread to suspend and migrate to a remote machine to continue its execution. Computations are migrated to remote locations using a round-robin schedule.

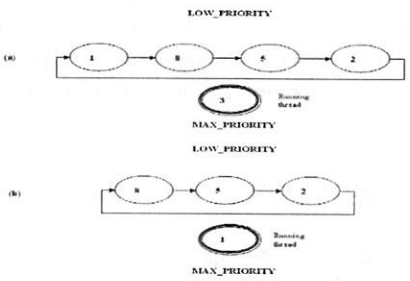


Figure 6. Thread Scheduler.

5.3.4 ClassLoader

In order to load classes from remote locations, we implemented our own classloader. The *BehvLoader* allows classes to be loaded over the network and stored within the local cache. The BehvLoader loads classes to the interpreter using the following sequence of operations (Figure 7).

1. Check if the class already exists in the local cache. If not,
2. Check if the class is a system class. If not,
3. Check the local disk. If not found,
4. Check the remote disk where the request originated. If not found,
5. *NoSuchClassFound* exception is thrown.

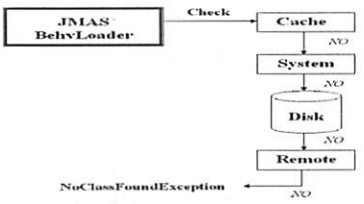


Figure 7. Operation of JMAS ClassLoader.

Different features can be added to the BehvLoader to provide security. Such as:

- encryption/decryption of class files
- use of signatures

5.3.5 Load Balancer

We implement a load balancing scheme based on the CPU market strategy proposed in [12]. The market strategy is based on CPU-time. Entities within the system consist of *buyers* and *sellers*. A *seller* allows its CPU to be used by other programs. A *buyer* serves as a machine wanting to offload work to a seller. A meeting place in which buyers and sellers are correlated is known as a *market*. CPUs are chosen from the market using three selection policies:

1. Optimal (Best) selection,
2. Round-Robin selection, or
3. Random selection.

5.3.5.1 Developing a Market of CPUs

We implement a decentralized hierarchical method for organizing the CPU market. Each machine within the system is responsible for managing a market. Therefore, the process of managing a market is distributed throughout the system–increasing market reliability and availability. When starting the system, the D-RTM initializes its market by registering itself with machines designated within a configuration file set by the system administrator. Those machines willing to sell their CPU respond with a message *SELLER*, and are added to the market as *sellers*. Machines who wish to buy CPU time respond with a message *BUYER*, and are added to the market as *buyers*. Those who do not respond (i.e. system down) are not added to the market. This market maintained by the D-RTM, contains the secondary machines on which to off-load remote processes. As shown in Figure 8, this creates a logical hierarchy of machines. Each node within the hierarchy, with the exception of the bottom most nodes, are denoted as market managers. Communication overhead is minimal. CPUs wishing to sell their time add themselves to the market by notifying a market manager (Figure 8). Buying from the market is a bottom up process. Nodes at the lowest level become overloaded faster. Once a given node X is denoted as a buyer, all nodes who are descendants of X are also denoted buyers. This approach requires collaboration among system administrators to organize an optimal hierarchy. This is not suitable for a global environment which must scale to hundreds or thousands of machines.

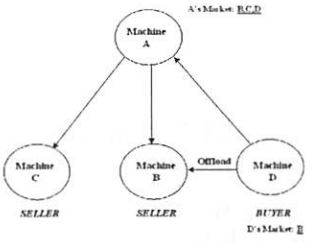


Figure 8. CPU Market Hierarchy.

We modify the hierarchical method, by allowing market initialization and registration to be bi-directional. Not only does the D-RTM register itself with machines designated by the system administrator, but machine also registers itself with the D-RTM. In such a situation, the market is organized by managers who are logically connected in a (complete) multidirectional topology. Because machines belong to

more than one market, this configuration increases the communication overhead substantially. Communication increased from one message round to an expensive multicast. As shown in Figure 9, not only do machines B, C, and D notify machine A when buying or selling their CPU time, but, machine A must also notify machines B,C, and D when buying or selling its CPU time. Changes in the CPU status (i.e. Buyer/Seller), are notified to all machines within a market using a weak consistent replication strategy. We use weak consistent replication in order to reduce the communication over head. Notifications are replicated throughout the system by piggybacking the CPU status of the current machine along with communication sends. For example: when an actor on machine B receives a communication from and actor on machine A, the CPU market on machine B is updated with the new CPU status of machine A. Although, machines are not instantly notified of a market change, use of this weak replication strategy provide eventual message delivery that is tolerated in our system [11].

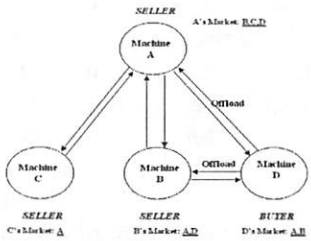


Figure 9. Host A Notifies Markets of B,C, and D.

5.3.5.2 Load Balancing Policy

Each machine within the distributed system maintains a data structure with information about the current machines within its market. These machines are denoted as buyers, or sellers. The load factor on the machine is relative to the number of threads currently running on the local machine. Other factors could also be used to determine the load. Such as: the total load on the machine, heuristic information, the actual CPU utilization, and the size of the computation. Most of these metrics are more complicated to determine. As shown in Figure 10, the Load Balancer maintains a load below 75% of the threshold, and 25% of the threshold above the minimum load (i.e. zero).

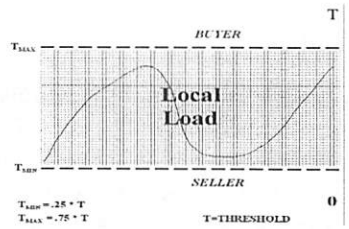


Figure 10. Load Balancing Policy.

Before starting a thread on the local machine, the load balancer checks the current load to insure that it is within the threshold. If the load is not within the current threshold, the load balancer off-loads a local process to machines within its market who wish to sell their CPU (Figure 11). If there are no sellers within the market, the load balancer starts the process locally, and tries to off-load processes later. Note that the D-RTM is now a buyer of CPU time and needs to inform its market managers of its new status. By default the status of a machine is *seller*. Therefore this field is changed to status *buyer*.

Variable Definitions:

t : Task (communication sent throughout system)

load : Integer to denote the current load on the local machine

Threshold : Integer to denote the load limit on the local machine

BUYER,SELLER : constant to denote the state of the machine

CPUStatus : enumerator to denote the state of the machine (BUYER/SELLER)

host : contains the host location of an available CPU

scheduleLocal(t) : schedules the Task t (i.e. an actor) on the local machine

scheduleRemote(t, *host*) : schedules the Task t (i.e. an actor) at the location *host*

getAvailHost() : returns an available CPU (SELLER) from the market,

updateMarket(t) : update the *CPUStatus* of the machine from which the Task t originated

LoadBalancer :

1. Receive Task t
2. If t is an actor

   if $load + 1 \leq Threshold$, then

   set *CPUStatus* to **SELLER**
   scheduleLocal(t)
   increment *load* by 1

   Else

   set *CPUStatus* to **BUYER**
   host = getAvailHost()
   scheduleRemote(t, *host*)

   Else if t is a communication

   updateMarket(t)
   forward task to Message Handler

3. goto 1

Figure 11. Load Balancing Algorithm.

5.4 Logical Layer

The logical layer consists of the actual application specific computations that are executing on the local machines. The computation model consists of mobile actors which encapsulate: a behavior, an identity, a mail queue, and one thread. Each computation runs in its own thread, and may communicate with any other thread on the local/remote machines. Computations are expressed as Java programs using mobile actor semantics provided by constructs of the JMAS Mobile Actor API. The mobile actor API gives programmers the ability to create actors, change the state, or send communications to mobile actors within the global system. The underlying resources can be logically represented as mobile actors to build dynamic architecture topologies. This dynamic architecture gives the programmer an illusion of a global computer that can run concurrent, distributed, and parallel applications. Implementation details of the underlying system are transparent to the programmer in the logical layer.

6 Performance Evaluation

JMAS offers the basic infrastructure needed to integrate computers connected by a network into a distributed computational resource: an infrastructure for running coarse-grain parallel applications on several anonymous machines. Currently, cluster computing in a LAN setting are already being used extensively to run computation intensive applications [17],[19]. We conducted our experiments in an environment consisting of:

- 1 Sun MicroSystems Enterprise 3000, configured with two UltraSparc processors each running at 256MHz.
- 1 Sun Ultra Sparc workstations, configured with one 120 MHz processor.
- 14 Sun Sparc 20 workstations, each configured with one 200 MHz processor.
- 1 Sun Sparc 10 workstations, configured with one 166 MHz processor.

Each machine is connected by a 10 and 100 Mbit Ethernet. All experiments were conducted under the typical daily workloads. We tested each algorithm under a controlled environment of D-RTMs that were used strictly to run our experiments. CPU selection from the CPU market, was performed by the D-RTM using a round-robin selection policy. Under our controlled environment, an optimal selection policy achieves the same results as round-robin CPU selection. We did not run our experiments using a random CPU selection policy. This was done to insure that all processes mapped to one and only one machine. In order to obtain a relative performance of our system, we calculate the average of the execution times over $N = 10$ experiments, producing an arithmetic mean (AM):

$$AM = \frac{1}{N} \sum_{1}^{N} Time_i$$

Where $Time_i$ is the execution time for the ith experiment. All experiments are compared with performance metrics obtained from similar computations on stand-alone workstations.

6.1 Benchmarks

The overhead of migrating actors to remote locations and passing messages between remote actors are of great interest. We present experimental results for our prototype using two benchmarks: a Traveling Salesman application, and a Mersenne Prime application. We discuss their implementation and performance using the JMAS infrastructure.

6.2 Factors That Limit Speedup

A number of factors can contribute to limit the speedup achievable by a parallel algorithm executing in a network computing infrastructure such as JMAS. An obvious constraint is the size of the input program. If there is not enough work to be done by the number of processors available, then any parallel algorithm will not show an increase in speedup. Second, the number of process creations must be minimized. In particular, we are concerned with the creation of remote actors throughout the distributed system. Lastly, in a network computing environment were communication cost is high, the number and packet size of inter-process communications must be limited. Table 1 shows the performance of two micro-benchmarks to calculate the execution time for communication sends, and remote class loading using the JMAS prototype. A micro-benchmark is a small experiment used to monitor the performance of underlying system operations. Results were obtained using a test packet to send a communication, and load a Java class file between two machines.

Overhead	secs
Send	.006-.010
Remote Class Loading	.15-.28

Table 1. Micro benchmarks for a 10 Mbit Ethernet LAN using TCP sockets.

In general, the total cost of distributing a program for parallel execution is defined as:

$$T_{Cost} = Total_{loadTime} + Total_{commTime} + Total_{execTime}$$

Where $Total_{loadTime}$ is the time to load the needed Java class files to each machine within the system, $Total_{commTime}$ is the time spent sending communications between actors, and $Total_{execTime}$ is the time spent executing the fraction of the computation. Moreover, the total time to distribute the needed Java class files across N machines is:

$$Total_{loadTime} = (N-1) * t_{load}$$

Where t_{load} is the average time to load the needed Java class files to one machine within the system. We assume that the machines are organized using a master-slave topology. Such that, the master is used to process a subcomputation, as well as, distribute $N-1$ subcomputations and receive the partial results from the other $N-1$ slave machines. Assuming we distribute the load evenly among N machines. Then the time to execute a fraction of the computation is:

$$Total_{execTime} = t_{seq}/N$$

Where t_{seq} is the total sequential execution time for the application. Given the load distribution above, if each subcomputation sends at most k messages, then the communication overhead $Total_{commTime}$ can be defined as:

$$Total_{commTime} = (N-1) * k * t_{send}$$

Where t_{send} is the average time to send a communication between two machines. Given N machines, we derive a general formula to define the total cost of distributing a program for parallel execution.

$$T_{Cost}(N) = (N-1) * t_{load} + (N-1) * k * t_{send} + t_{seq}/N$$

Using the equation above, we can estimate the performance of a given application. As shown below, in order to benefit from parallelization the following inequality must hold:

$$T_{Cost}(N) < t_{seq}$$

$$(N-1) * t_{load} + (N-1) * k * t_{send} + t_{seq}/N < t_{seq}$$

Solving the inequality, we find that the total cost (i.e. $T_{Cost}(N)$) is less than the sequential execution time (i.e. t_{seq}) for:

$$N < t_{seq}/(t_{load} + k * t_{send})$$

6.2.1 Remote Execution of Actors

As a mobile actor computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network; causing the migration of code. On each of the experiments conducted in this chapter, we calculated the average time to load a Java class file over the network. On a standard 10 Mbit Ethernet network the time to load a remote class file ranges between .15 and .28 seconds (Table 1). On average it takes .20 seconds to load a class file across the network. When considering distributing an application across several machines, one must take into consideration an upper bound on the amount of parallelism that can be exploited by distributing processes throughout a network computing system. In particular, we focus on the overhead associated with loading Java class files across the network (i.e. $Total_{loadTime}$). We can calculate the maximum number of machines p, needed to distribute the parallel computation without compromising the performance in speedup by finding the absolute minimum execution time for the continuous function $T_{Cost}(p)$ on a closed bounded interval $[1, p]$; where $p = t_{seq}/(t_{load} + k * t_{send})$. Giving,

$$T'_{Cost}(p) = t_{load} + k * t_{send} - t_{seq}/p^2$$

Setting $T'_{Cost}(p) = 0$ and solving for p, gives

$$p = \sqrt{t_{seq}/(t_{load} + k * t_{send})}$$

Therefore, we can estimate the maximum performance in speedup S as:

$$S = t_{seq}/T_{Cost}(p)$$

$$T_{Cost}(p) = 2 * t_{load}\sqrt{t_{seq}/(t_{load} + k * t_{send})} - t_{load}$$

Giving,

$$S = \frac{2 * t_{seq}\sqrt{t_{seq}/(t_{load} + k * t_{send})} + t_{seq}}{4 * t_{seq} - (t_{load} + k * t_{send})}$$

6.2.2 Message Passing

As stated in Chapter 5, communication in JMAS is asynchronous, reliable and connection-oriented. Messages between two actors, must be routed through a

D-RTM on the local machine on which the two actors reside. The Java Virtual Machine requires all communication to go through the Java network layer (i.e. *java.net*) and the complete TCP stack of the underlying OS. This causes a substantial software overhead compared to communication libraries of parallel machines. Using JMAS, a single message can be sent from one actor to another within .006-.010 seconds on a standard 10 Mbit Ethernet LAN (Table 1). As long as applications are coarse grained, the overhead of opening a socket connection can be ignored. Since message passing using Java TCP sockets is slow compared to dedicated parallel machines, and communication delays of large networks of heterogeneous machines is unpredictable, only computation-intensive parallel applications benefit from the JMAS infrastructure.

6.3 Traveling Salesman Problem

Our first application is a parallel solution to the Traveling Salesman Problem (TSP). The Traveling Salesman Problem is as follows: given a list of n cities along with the distances between each pair of cities. The goal is to find a tour which starts at the first city, visits each city exactly once and returns to the first city, such that the distance traveled is as small as possible. This problem is known to be NP-complete (i.e. no serial algorithm exists that runs in time polynomial in n, only in time exponential in n), and it is widely believed that no polynomial time algorithm exists. In practice, we want to compute an approximate solution, i.e. a single tour whose length is as short as possible, in a given amount of time.

6.4 TSP Algorithm

We take a naive approach to solving the TSP using an Exhaustive-Search. The exhaustive-search algorithm searches all $(n-1)!$ possible paths, while keeping the best path searched so far. We generate all possible paths using a $Perm()$ function on the number of cities n. The permutation function generates a lexicographical ordering of all possible paths. We divide the permutations equally among a set of processors p; such that each processor searches $(n-1)!/p$ possible paths (Figure 12). Processors are arranged in a master-slave design.

Variable Definitions:

- *n* : Integer to denote the number of cities
- *p* : Integer to denote the number of machines
- *mintour* : Integer to denote the permutation of the best tour searched
- *start* : Integer to denote the starting permutation in lexicographical order
- *stop* : Integer to denote the ending permutation in lexicographical order
- *resultTour* : Integer to denote the best tour search for a specified range lexicographically
- *itself* : Actor address of itself
- *cust* : Actor address to send result
- *range* : Integer to denote the total permutations (tours) to check
- Perm(*i*) : Generates the *i*th tour in lexicographical order

behavior Slave :

1. recv *start*, *stop*, and address of *cust* to send result
2. *mintour* = *start*
3. for *i* equal *start* to *stop* do

   if Perm(*i*) distance $\leq$ Perm(*mintour*) distance
   set *mintour* to *i*
4. **send** *mintour* to *cust*

behavior Master :

1. *mintour* = 0
2. *range* = $(n-1)!/p$
3. for each processor *i* : 1 to $p-1$ do

   create a Remote actor assume behavior *Slave*, return address of actor as *x*
   send *start* = *(i*range)*, *stop* = *((i+1)*range)*, and the address of *itself* to *x*
4. **become** *itself* and wait for *p* results
5. for *i* : 1 to *p* do

   receive *resultTour*
   if Perm(*resultTour*) distance $\leq$ Perm(*mintour*) distance
   set *mintour* to *resultTour*

Figure 13. TSP Algorithm.

6.4.1 Measurements

In order to complete our set of measurements in a reasonable amount of time we chose to test our TSP solution primality for $N = \{4, 5, 10, 13\}$ cities. We conducted the experiment in an environment consisting of up to 15 machines, and compared the results with a sequential application running on a SPARC 20 workstation. As shown in Figure 13, there is no significant gain in performance for $N < 10$. This is due to the overhead associated with loading Java class files across the network. Figure 14 displays the execution time of a TSP solution for $N = 5$ versus its remote Java class loading time. As the number of machines p increase, the load time increases, causing the execution time to increase; exceeding the execution time for a sequential solution. Notice we achieve the best

performance for $p = 4$ machines. For $N \geq 10$, our TSP solution gives a better performance. In particular, for $N = 13$ the speedup obtained is close to linear. Due to limited resources, we were unable to test the scalability of the application for large values of p. We estimate the performance of our TSP application using Equations 1,2 and an average load time $t_{load} = .15$ secs. As illustrated in Table 2, the average CPU utilization for the best possible number of machines p is 50%. This is because, as the number of processors p approach $(N-1)!$, the speedup obtained will decrease significantly; due to under utilization of processors and the overhead associated with loading Java class files across the network (Figure 15). The estimates are also reflected in Figure 13. These results show that our framework is well suited for course grain applications. The TSP application also scales well to large computation sizes (Figure 16).

Prob. Size	t_{seq} secs	Max. p	Max. S	Utilization
N=5 Cities	3.007	4	2.24	56%
N=10 Cities	24.441	12	6.33	52.7%
N=13 Cities	36655.848	494	247.42	50%

Table 2. Estimating the Performance of TSP.

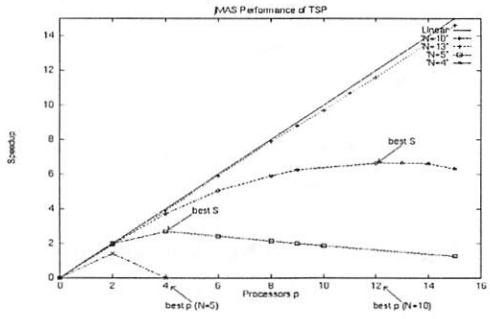


Figure 13. Speedup of TSP.

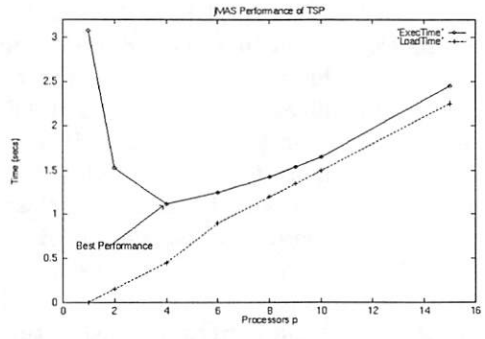


Figure 14. Execution Time vs Load Time.

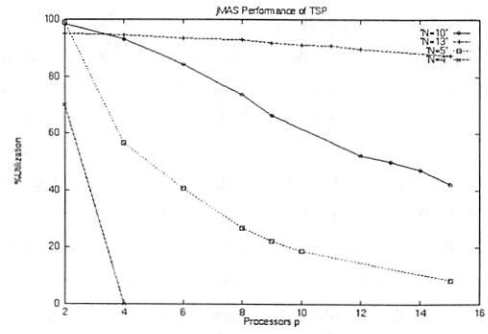


Figure 15. CPU Utilization of TSP.

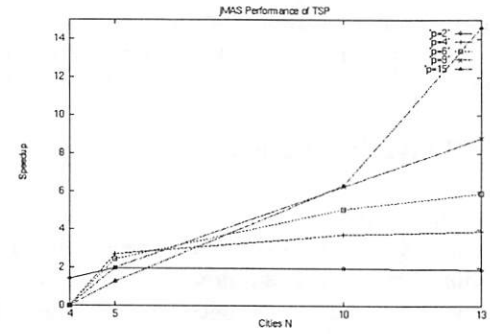


Figure 16. Scalability of TSP.

6.5 Mersenne Prime Application

For our second application, we implemented a parallel primality test which is used to search for Mersenne prime numbers [19]. This type of application is well suited for our infrastructure. It is very coarse grained with low communication overhead.

A Mersenne prime is a prime number of the form $2^p - 1$, where the exponent p itself is prime. These are traditionally the largest known primes. Encryption and decryption methods are typical applications which utilize large prime numbers. Searching and verifying Mersenne primes using computer technology has been conducted since 1952 [19]. To date 37 Mersenne primes have been discovered. Only up to the 35th Mersenne prime has been verified. The current record holder is $2^{1398269} - 1$ and was discovered through the use of over 700 PCs and workstations worldwide. With larger and larger prime exponents, the search for Mersenne primes becomes progressively more difficult.

6.5.1 Mersenne Prime Algorithm

In our implementation, each prime is tested based on the following theorem:

Lucas-Lehmer Test: For p odd, the Mersenne number $2^p - 1$ is prime iff $2^p - 1$ divides $S(p-1)$; where $S(n+1) = S(n)^2 - 2$, and $S(1) = 4$. The proof can be obtained from [19].

We develop a mobile actor program to test for Mersenne primality, given a range of prime numbers (Figure 18). Processors are arranged in a master-slave design. As shown below, our application works as follows:

> Given N machines and a range r of prime numbers, we divide the search such that each machine tests for a Mersenne prime using the Lucas-Lehmer Test for a range of primes. Each range is of size r/N.

Variable Definitions:

r : Integer to denote the amount of primes to test
N : Integer to denote the number of machines
Lucas(x) : Performs Lucas-Lehmer test on x
itself : Actor address of itself
cust : Actor address to send result
range : Integer to denote the range of primes to check
start : Integer to denote the starting prime number
stop : Integer to denote the prime number used as a sentinel
$recv_{count}$: Integer to denote the total results received
PRIME : enumerator returned from Lucas(x); if x is a prime number
SINK : message to denote the termination of a subcomputation

behavior Slave :

1. recv *start*, *stop*, and address of *cust* to send result
2. for i : *start* to *stop* do
   if Lucas(i) is **PRIME**
   send i to *cust*
3. **send** *SINK* to *cust*

behavior Master :

1. $range = r/N$
2. for each processor i : 1 to $N-1$ do
   create a Remote actor assume behavior *Slave*, return address of actor as x
   send *start* = *(i*range)*, *stop* = *((i+1)*range)*, and the address of *itself* to x
3. **become** *itself* and wait for N results
4. set $recv_{count} = 0$
5. receive result
6. if *result* is *SINK*
   increment $recv_{count}$ by 1
   Else
   print "$2^{result} - 1$ is PRIME!"
7. if $recv_{count} < N$, then goto 5

Figure 17. Mersenne Prime Algorithm.

6.5.2 Measurements

For our measurements, we chose to test the Mersenne primality for all exponents between 4000 and 5000. Known primes within this range are $2^{4253} - 1$ and $2^{4423} - 1$. The reason for selecting this range is that:

1. we tried to make the number large enough to simulate the true working conditions of the application,
2. we wanted to keep them small enough to be able to complete our set of measurements in a reasonable amount of time.

We conducted the experiment in an environment consisting of up to 15 machines, and compared the results with a sequential application running on a SPARC 20 workstation. As shown in Figure 18, our application scales to 15 machines linearly. The speedup obtained is slightly lower than linear speedup. This is because we decompose the range of primes to be tested unevenly in terms of the amount of work to be done.

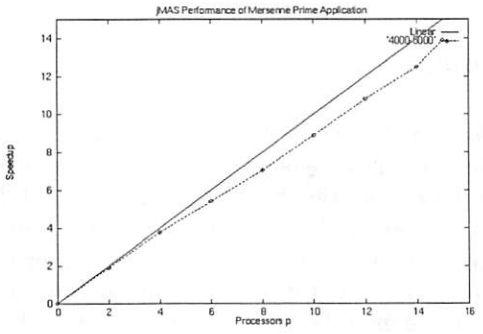


Figure 18. Speedup of Mersenne Prime.

For instance, testing if $2^{4000} - 1$ is prime, can be done much faster than testing if $2^{4999} - 1$ is prime. We split the ranges in groups such that, the last machine receives the last group consisting of the largest numbers. Due to limited resources, we were unable to test the scalability of the application for large values of p. We estimate the performance of the Mersenne Prime application using Equations 1,2; where the average load time $t_{load} = .20$ secs, and the average sequential execution time $t_{seq} = 83432$ secs. As shown in Table 3, results show that the application scales up to 646 machines with an overall speedup of 323. From our results we can assume that for $p > 646$, the range of primes to test decreases causing under utilization of CPUs (Figure 19). Also, for every new machine added, the time to load Java class files increases causing a decrease in performance.

Application	t_{seq} secs	Max. p	Max. S	Utilization
Mersenne Prime	83432	323	646	50%

Table 3. Estimating the Performance of the Mersenne Prime Test.

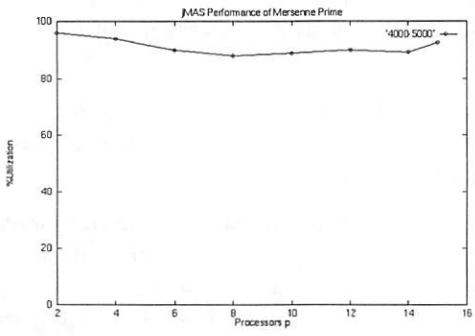


Figure 19. CPU Utilization of Mersenne Prime.

7 Conclusion

In this paper we discuss the design and implementation of a prototype network computing system (*JMAS*) based on the mobile actor model [10] using Java technology [25]. JMAS requires a programming style different from commonly used approaches to distributed computing. JMAS allows a programmer to create mobile actors, initialize their behaviors, and send them messages using constructs provided by the JMAS Mobile Actor API. As the computation unfolds, mobile actors have the ability to implicitly navigate autonomously throughout the underlying network. New messages are generated, new actors are created, and existing actors undergo state change. We evaluate the performance of our system using two benchmarks: a Mersenne Prime Application, and the Traveling Salesman Problem. The degree of parallelism obtained from distributing mobile actors throughout the system is limited due to the overhead associated with migrating Java class files, and the amount of inter-process communication. In particular, we are bound by the number of processors

$$p = \mathbf{O}(\lfloor\sqrt{t_{seq}/(t_{load} + k * t_{send})}\rfloor)$$

to distribute the parallel computation; where t_{seq} is the sequential execution time of the application, t_{load} is the average time to load the needed Java class files to one machine, k is the total message rounds sent per machine, and t_{send} is the average time to send a communication between two machines. Given p we can estimate the speedup S as:

$$S = t_{seq}/T_{Cost}(p)$$

Where the enhanced performance using p machines, is denoted as a general formula

$$T_{Cost}(p) = (p-1) * t_{load} + (p-1) * k * t_{send} + t_{seq}/N$$

Our estimates for the TSP and Mersenne Prime applications, show that each application scales to large numbers of machines N. But for $N > p$, we estimate a decrease in performance; due to the under utilization of CPUs, and the significant overhead associated with loading the needed Java class files and sending communications throughout the system. These results show that our framework is well suited for course grain applications.

7.1 Future Work

Issues such as fault tolerance and security need to be addressed and implemented within the JMAS framework. Also, experiments concerning the scalability of the JMAS framework to support internet (global) computing will be conducted in future work. Support for high-level communication abstractions will be addressed within the JMAS Mobile Actor API. Examples are barrier actors, mutex actors, call/return communication, and actorSpaces [2].

References

[1] Gul Agha, Chris Houck, and Rajendra Panwar. Distributed execution of actor programs. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing.* Santa Clara, 1991.

[2] Gul Agha and R. Panwar. An actor-based framework for heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 21, 1991.

[3] T. Anderson, D. Culler, and D. Patterson. A case for now (network of workstations). In *IEEE Microcomputer.* IEEE, 1995.

[4] NPAC at Syracuse University. *WebFlow: A Visual Programming Paradigm for Web and Java Based Coarse Grain Distributed Computing.* Online Technical Report, http://www.npac.syr.edu/projects/javaforcse/cpande/sufurm.ps, 1997.

[5] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for WorldWide Applications.* ACM SIGOPS, 1996.

[6] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems.* PDCS, 1996.

[7] T. Berners-Lee. Www: Past, present and future. *IEEE Computer*, 18:69–77, 1996.

[8] L. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 18:55–61, 1996.

[9] R. Brecht, H. Sandhu, M. Shan, and J. Talbot. Paraweb: Towards world-wide supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for WorldWide Applications.* ACM SIGOPS, 1996.

[10] L. Burge and K. George. An actor based framework for distributed mobile computation. In *PDPTA - Parallel Distributed Processing Techniques and Applications.* CSREA, 1998.

[11] L. Burge and M. Neilsen. Variable-rate timestamped anti-entropy. In *ISMM International Conference on Parallel and Distributed Computing and Systems.* 7th IASTED, 1995.

[12] N. Camiel, S. London, N. Nisan, and O. Regen. The popcorn project: Distributed computation over the internet in java. In *Proceedings of the 5th Internation World Wide Web Conference.* W3, 1997.

[13] K. Chandy, B. Dimitron, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P. Sivilotti, W. Tawaka, and L. Weisman. A world-wide distributed system using java and the internet. In *Proceedings of the 5th IEEE Internation Symposium on High Performance Distributed Computing.* IEEE HPDCS, 1996.

[14] Emory University Dept. of Computer Science. *IceT: Distributed Computing and Java.* Online Technical Report, http://www.mathcs.emory.edu/ gray/, 1997.

[15] Old Dominion University Dept. of Computer Science. *Web Based Framework for Distributed Computing.* Online Technical Report, http://www.cs.odu.edu/ techrep/techreports/TR_97_21.ps.Z, 1997.

[16] University of California at Santa Barbara Dept. of Computer Science. *Javalin: Internet-Based Parallel Computing Using Java.* Online Technical Report, http://www.cs.ucsb.edu/ danielw/Papers/wjsec97.ps, 1996.

[17] DESCHALL. Internet-linked computers challenge data encryption standard. Technical report, Press Release, 1997.

[18] Online Document. *Mobile Agents: Are they a good idea?* http://www.eit.com/goodies/list/www.lists/ www-talk.1995q1/0764.html, 1995.

[19] Online Document. *Mersenne Primes: History, Theorems and Lists.* http://www.utm.edu/research/ primes/mersenne.shtml, 1998.

[20] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1, 1997.

[21] G. Fox and W. Formaski. Towards web/java based high performance distributed computing - and evolving virtual machine. In *Proceedings of the 5th IEEE Internation Symposium on High Performance Distributed Computing.* IEEE HPDCS, 1996.

[22] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *Proceedings of the 6th ACM International Conference on Supercomputing.* ACM, 1992.

[23] A. Grimshaw, W. Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 20:39–45, 1997.

[24] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[25] Sun Microsystems Inc. *The Java Virtual Machine Specification.* Online Technical Report, http://java.sun.com, 1995.

[26] L. Kale', M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel java with global object space. In *PDPTA International Conference*, pages 235–244. PDPTA, 1997.

[27] A. Keren and Institute of Computer Science Hebrew University A. Barak. *Parallel Java Agents.* http://cs.huji.ac.il/, 1998.

[28] Argonne National Laboratory and USC Information Science Institute. *The Nexus Multithreaded Runtime System.* http://www.mcs.anl.gov/nexus, 1997.

[29] M. Litzkow and M. Linwy. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems.* ICDCS, 1988.

[30] F. Reynolds. Evolving an operating system for the web. *IEEE Computer*, 1:90–92, 1997.

[31] V. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2, 1990.

[32] H. Takagi, S. Matsuoka, and H. Nakada. *Ninflet: A Migratable Parallel Object Framework using Java.* http://ninf.etl.go.jp/, 1998.

[33] L. Vanhelsuwe. *Create your own supercomputer with Java.* http://www.javaworld.com/javaworld/jw-01-1997/jw-01-dampp.html, 1997.

Adaptation and Specialization for High Performance Mobile Agents

Dong Zhou and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{zhou,schwan}@cc.gatech.edu

Abstract

Mobile agents as a new design paradigm for distributed computing potentially permit network applications to operate across dynamic and heterogeneous systems and networks. Agent computing, however, is subject to inefficiencies. Namely, due to the heterogeneous nature of the environments in which agents are executed, agent-based programs must rely on underlying agent systems to mask some of those complexities by using system-wide, uniform representations of agent code and data and by 'hiding' the volatility in agents' 'spatial' relationships.

This paper explores runtime adaptation and agent specialization for improving the performance of agent-based programs. Our general aim is to enable programmers to employ these techniques to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. The specific results in this paper demonstrate the beneficial effects of agent adaptation both for a single mobile agent and for several cooperating agents, using the adaptation techniques of agent morphing *and* agent fusion. *Experimental results are attained with two sample high performance distributed applications, derived from the scientific domain and from sensor-based codes, respectively.*

1 Introduction

Mobile agent[4, 14, 18, 42] as a new design paradigm for distributed computing potentially permit network applications to operate across heterogeneous systems and dynamic network connectivities, to reduce their bandwidth needs, and to avoid overheads caused by large communication latencies. In addition, mobile agent systems[11, 20, 24, 38] are designed to facilitate the construction of distributed programs that have the flexibility to adapt their operation in response to the heterogeneous nature of or dynamic changes in underlying distributed computing platforms.

Agent computing, however, is subject to several inefficiencies. Some of these inefficiencies are caused by the complexities of the environments in which mobile agents are deployed. Such environmental complexities include heterogeneity in architectures, communication networks (both at the hardware and protocol levels), operating systems, and agent management systems. This diversity requires agent-based programs to rely on underlying agent systems, most of which are based on interpreted languages like Java and Tcl/Tk[10, 26], to mask some of these complexities, by using system-wide, uniform representations of agent code and states to store, transport and execute agent programs. Additional inefficiencies in agent computing are caused by the dynamic nature of agent-based programs, where different components of these programs exhibit volatile 'spatial' relationships. Such 'spatial' volatility results from agents' mobility and from the runtime service/agent discovery schemes being used. The underlying agent systems 'hide' this volatility by ensuring that remote agent invocations are directed to current agent execution sites.

There has been considerable work on dealing with inefficiencies in agent computing, including the development of Just In Time (JIT) compil-

ers for agent code[23], of methods for creating efficient Java programs[37], and of performance tuning techniques and tools for distributed agent applications[15]. These efforts are particularly relevant to performance-constrained distributed applications, such as data mining, where large amount of states may have to be moved upon discovery, and interactive simulations, where application must offer real-time performance to end users [12, 21].

Our research is exploring two approaches for improving the performance of distributed, agent-based programs: (1) runtime adaptation and (2) agent specialization. The general aim of this work is to enable programmers to employ these techniques to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. This paper explores the effects of using two specialization approaches: *agent morphing* on a single mobile agent and *agent fusion* on multiple cooperating agents.

The remainder of this paper is organized as follows: Section 2 presents two applications that can benefit from the use of agent technologies while also requiring levels of performance not easily attained with current agent systems. interactive continuously Section 2 also describes the performance implications of using agent- vs. compiled object-based representations of the software components involved in interactive data viewing. Based on these evaluations, Section 3 next describes two agent specialization techniques – morphing and fusion – that address some of the runtime performance problems of applications like these. Significant performance gains are demonstrated from applying these techniques to the aforementioned applications for a variety of typical scenarios of use. Based on the improvements demonstrated in this section, Section 4 then describes next steps in our research, including the design of runtime support in which various adaptation techniques are easily applied. The paper concludes with a discussion of related research (see Section 5), conclusions, and future work (Section 6).

2 Using Agents with High Performance Applications

2.1 Mobile Agents and High Performance

Advances in the processing and communication capabilities of today's computer systems make it possible to wire heterogeneous and physically distributed systems into computational grids[8] that are able to run computation- and communication-intensive applications in real time. Consequently, end users are encouraged to interact with their applications while they are running, from simply inspecting their current operation, to 'steering' them into appropriate directions[27]. Examples of such applications include teleimmersion, interactively steered high performance computations, data mining, distributed interactive simulations, and smart sensors and instruments [12, 21, 41, 44].

Data in such applications comes from sources like sensors, disk archives, network interfaces, and other programs, is transformed while passing through the computational grid, and is finally output into sinks like actuators, storage devices, and the user interfaces employed by interactive end users. Application interfaces also permit applications to be reconfigured on-line in response to explicit user requests or to changes in user behavior. Sample reconfigurations include the creation or termination of certain application components, component replication, changes in dependencies between components, and changes in the mapping of components to computational grid elements.

Our aim is to use mobile agents to implement some of the data processing tasks of interactive high performance applications. More specifically, while it is unlikely that a high performance simulation like a fluid dynamics[39] or a finite element code will employ mobile agents for the simulation itself, it is desirable to represent as agents many of the computations and data transformations required for their interactive use. Such representations enable end users to interact with their long running simulations from diverse locations and machines (e.g., when working from home), and they permit the appropriate placement of data transformations such that data reductions are performed where most appropriate (e.g., before sending data to a weakly connected machine located in an end user's home). Our efforts are supported by several recent developments, including the

creation of agent-based visualization and collaboration tools for high performance computations[13, 40].

Our second aim is to freely mix the use of agent- vs. compiled object-based representations of data transformation tasks, such that end users need not be aware of the current task representations and such that changes in task location and representation are made in response to current user behavior and needs. This paper presents our design ideas and initial implementation concerning a mixed agent/object system. This work is based on recent work elsewhere on object or agent specialization[28] and by our own work on object technologies for high performance and interactive parallel or distributed programs[6, 35, 36].

The remainder of this section describes and evaluates two applications that are representative of interactive scientific programs and sensor processing applications, respectively. The application drawn from the scientific domain, termed Interactive Access to Scientific Data (ISDA), has performance constraints due to the amounts of data being manipulated and displayed, regardless of end users' locations. The other application's performance constraints are derived from the necessity to process data in real-time or at certain rates, while sensor (source) and sink locations may change. This application, termed Parallel Scalable SAR Processing Simulator (PSSPS) is derived from the standard SAR (Synthetic Aperture Radar) benchmark originally developed at Lincoln Labs[46].

2.2 Sample Applications: Interactive Scientific Data Access (ISDA) and Parallel Scalable SAR Processor Simulator (PSSPS)

Both the ISDA and PSSPS applications are stream-based, driven by multiple inputs (stream sources) and able to service any number of end points (stream sinks). The purpose of ISDA (Figure 1) is to enable human end users to view and steer a high performance simulation (a global atmospheric model acting as a data source) via visualizations of the model's output data (ie., the stream sinks). The model simulates the transport of chemical compounds through the atmosphere. It uses assimilated windfields derived from satellite observational data for its transport calculation, and known chemical concentrations also derived from observational data as the basis of its chemistry calculations.

Of interest to this paper are the ancillary computations that 'link' the atmospheric model itself to various visualizations, where the set of these additional computations is depicted as a 'cloud' connecting the simulation to its inputs/outputs in Figure 1. Most such 'cloud elements' implement transformations that prepare model data for viewing by end users, e.g., reducing the amount of model data, transforming model data from its model-internal to a user-viewable representation, etc. Other 'cloud elements' perform additional computations like comparing model outputs with satellite observational data.

The specific cloud elements used in our work are (1) a regression model with which statistical tests may be performed on selected data, (2) a specialized data reduction code that 'clusters' scientific data[29] as per end user needs, (3) the Spectral-to-Grid transformer(s) that transforms the simulation model's internal data representation to the grid-based representation suitable for data visualization, and (4) the Isosurface calculator(s) that computes volumes of data of interest to end users and also generates the graphical primitives based on which this data may be viewed (an example of data of interest to end user is data volumes in which certain chemical constituents have equal levels of concentration, and an example of generated graphical primitives are the triangular representations of data suitable for the OpenGL rendering commands used in the 3D data visualization).

For the ISDA application, the argument for using agent-based representations for selected 'cloud' elements is apparent from the fact that the visualization engines themselves may have agent-based representations, as in the case of VizAD[40], in addition to object-based representations such as the SGI OpenInventor-based visualizations described in [30]. Specifically, when end users work in laboratories, they are likely to use high end machines capable of running the OpenInventor-based visualizations interactively in order to inspect model data in detail. When end users are simply 'looking a colleague over the shoulder' with the collaborative interfaces used in our work, then they require less detailed information and are likely to use ubiquitously runnable visualization tools like VizAD that enable such collaboration across a large diversity of machines and locations.

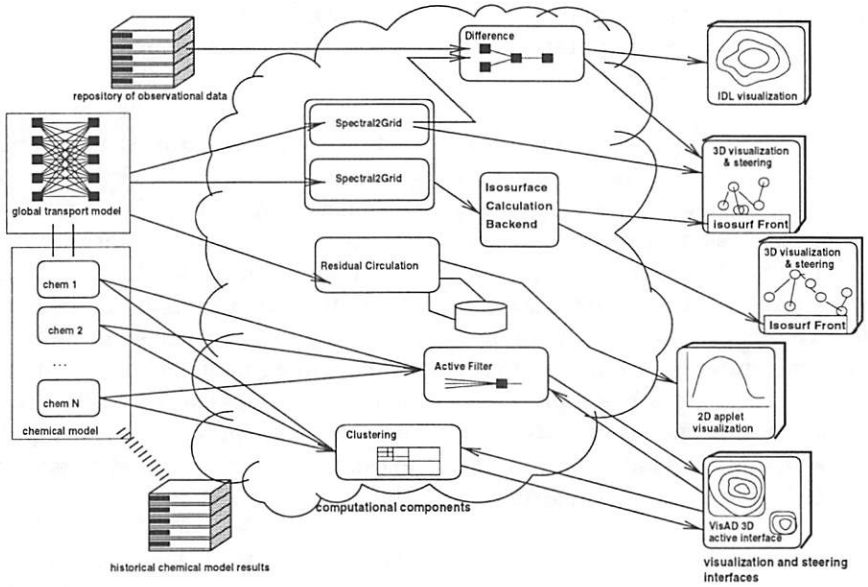


Figure 1: Computational components in ISDA.

Consequently, the associated data transformations may need to change data representations repeatedly, first from the model's internal spectral form of data to grid form, second from grid form to descriptions that may be rendered graphically, as exemplified by an Isosurface calculator. This transformer may reside as an agent on the same machine as the agent-based VizAD visualization or it may reside on a remote machine and operate as a specialized data reduction engine if the VizAD visualization is run on a weakly connected machine, such as a laptop or a computer located in a user's home. Clearly, the suitable choices of representation and the locations of agent- or object-based cloud elements depend on many factors, including current user needs and computing platform characteristics. The results presented below represent a first step toward automating choices like these, as they demonstrate the tradeoffs in performance when different element representations are used.

In PSSPS (Figure 2), data is either synthesized online or read from disk files that contain radar images. The processing of this data is performed by cloud components that include (1) selectors that filter out uninterested frames, (2) FIR filters for video to baseband I/Q conversion, (3) range compression units for pulse compression, and (4) Azimuth Compression Units for cross-range convolution filtering. Convolution results are the output strip-map images used for visualization. The implementation of PSSPS used in our work exhibits both the pipeline parallelism similar to that of the ISDA application and additional parallelism internal to pipeline stages that are able to utilize it, as is exemplified by the data parallel processing of the convolution stage for the purpose of speeding up this process.

In the PSSPS application, agent representations are useful for sensors in remote or mobile locations and/or for end users who wish to understand sensor data from mobile or remote locations. One example is a battlefield where radar data should be made accessible in some form to mobile units in the field operating at locations remote from the radar itself, only when such operational capabilities are currently required, whereas more permanent processing is installed and operated at regional or global command sites. This implies the need for flexibility in the location and execution of agent-based SAR computations associated with data sources and sinks.

2.3 Tradeoffs across Alternative Program Representations

The basic performance problems arising from the use of agent vs. compiled object representations of ISDA or PSSPS components are well understood. Usage of interpreted languages, such as Java[38, 20, 24], is a major cause of these problems as is depicted by experiment results listed in table 1. These ex-

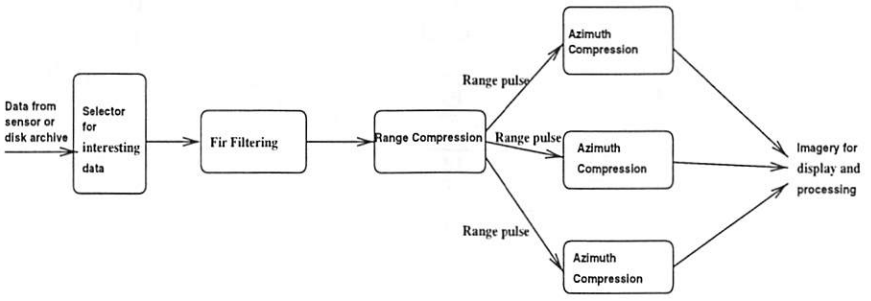


Figure 2: Structure of PSSPS.

periments use the Sun Solaris native C compiler to generate compiled code and use JDK1.2-beta3 package for the Java compiler and runtime environment (including the JIT compiler used in our later experiments). The platforms used in the experiments are the Sun Ultra-Sparc 30 uniprocessor systems with Solaris 2.5.1 and 100MB FastEthernet interconnection.

Specifically, these measurements demonstrate that for an application component like PSSPS' Azimuth computation, which has a large amount of computationally expensive floating point operations, the Java code realization runs almost 10 times slower than compiled code implemented with C. And similarly, for an application component like ISDA's Isosurface generation back-end, the Java code implementation on average takes 17 times more time than its native counterpart for a random set of grid data.

Agents	java code	compiled code	ratio
Azimuth processing	30,992	2,948	10.513
Isosurface back-end	8,348	461	18.11

Table 1: Comparing the performance of Java vs. native code(in msecs except for 'ratio').

In comparison with Table 1, the measurements presented next demonstrate the utility of JIT compilers for Java, which constitutes one way in which agent-based programs and their runtime environment may be specialized for efficient execution on target machines that have such compilers available. Specifically, Table 2 shows that with JIT, Java realizations of application components are from more than 3 times (for Isosurface back-end) to close to 5 times (for Azimuth processing) faster than those without JIT. This table also shows that static optimizations done by compilers vary in their effectiveness and that Java inter-class optimization does not much affect either of the two application components.

The performance improvements demonstrated in Table 2 might be sufficient for some applications. However, for applications like PSSPS and ISDA, their scalability and utility for large-scale data sets and for realistic execution rates would be compromised substantially by the fact that their JIT-based Java representations are 70% (in the case of Azimuth processing) to 300% (in the case of Isosurface back-end) slower than native code. However, perhaps even more important is the fact that significant additional overheads exist for distributed agent-based programs in which multiple agents must cooperate remotely, as is the case for both the ISDA and PSSPS applications. Agent-based high performance systems obviously need efficient communication mechanism to facilitate cooperation, which may involve large amounts of data, among agents. Unfortunately, our third experiment shows that Java RMI[43], which is being used for agent communication in many of the Java-based agent systems, has overheads which limit these applications' scalability in the presence of intensive agent communication.

Our experiment uses three components of PSSPS: Fir filtering, Range processing and Azimuth processing, to construct three pipelines with length of 0(Azimuth only), 1(Azimuth and Range), and 2(all three components). Our agent implementation uses Java and RMI, while the compiled object implementation uses C and OTL. OTL is the object invocation layer of the COBS CORBA-compliant object infrastructure developed at GT for high performance object-based programs[36, 7]. OTL is built on top of TCP and can perform object invocation across heterogeneous platforms.

Agents		No Optimization	Normal Optimization	Inter-class Optimization
Azimuth processing	Without JIT	30,992	30,885	30,736
	With JIT	5,246	5,223	5,258
	Ratio	5.908	5.913	5.846
Isosurface back-end	Without JIT	8,348	7,452	7,469
	With JIT	1,921	1,812	1,819
	Ratio	4.364	4.113	4.106

Table 2: Effects of using JIT compilation(in msecs except for 'ratio').

The experimental results depicted in Table 3 demonstrate the importance of RMI performance for even the computationally intensive applications. With increased pipeline lengths, the relative performance of the compiler optimized, JIT-enabled Java representation over that of the compiled code using our efficient distributed object infrastructure gets progressively worse. We suspect that this performance problem of Java RMI is due to marshaling overhead (object creation, stream creation) and to threading and synchronization costs. There has been research on improving the performance of Java RMIs[19], but this has not yet resulted in improvements to the standard Java distribution.

pipeline length	java code	native code	ratio ratio
0	5,223	2,948	1.772
1	23,095	4,968	4.649
2	34,404	6,054	5.683

Table 3: JIT's effectiveness for pipelined applications(in msecs except for 'ratio').

The measurements depicted above demonstrate the need for additional optimizations of agent-based representations of distributed programs if they are to be used to implement high performance applications. One basic issue, we believe, is that current JIT-based optimizations lack information about the operation and behavior of these distributed applications that can be exploited to further improve their performance. Specifically, first, if the duration of an agent's operation is known, then it becomes feasible to *morph* at runtime an agent-based representation to one using native code, invisibly to end users and using techniques like cross-platform binary code generation or access to code repositories. The technique assumed in this paper relies on the presence of code repositories[2]. Second, if multiple agents residing and cooperating on one machine could be 'compiled' as if they were one unit, then this compilation could use global knowledge not accessible to either method-based JIT compilation or component-based Javac compile-time optimization, thereby able to address both intra- and inter-component (e.g., RMIs) performance issues. Such global properties are considered by the *agent fusion* technique explored in this paper, which combines multiple agents into single, more powerful and potentially, more efficient agent representations.

Morphing, fusion, and their application to the ISDA and PSSPS distributed programs are discussed in more detail next.

3 Techniques for High Performance Agent Realization

3.1 Agent Morphing

Morphing Concepts. One specialization mechanism proposed in this paper is morphing, which means changing the form of a mobile agent to adapt to the specific platform on which it is currently running. Namely, each agent may have two forms: a platform independent form – henceforth termed *neutral* form – and a platform dependent form – henceforth termed *native* form. The agent is programmed to be able to morph between these two forms, using some of the techniques exposed in Section 3 below.

This paper establishes the importance of agent morphing and describes the internal structure of morphable agents. Briefly, we assume that all agents start with their neutral forms, which implies that they have no architectural knowledge of the hosts when they are deployed; this also facilitates the dynamic introduction of new architectures into the

system. Once started, the agent will spawn a low priority thread to acquire its native implementation, and if such a native form exists, the agent can then switch from its former agent mode into native mode whenever deemed necessary. Such mode switching can occur either

- immediately after the agent has acquired its native form,
- when a morphing instruction in the program is encountered,
- or in response to end user request or to events generated by the quality of service management system[33] that controls agents/objects' efficient execution.

During mode switching, the system transforms and copies the agent-mode states into its native-mode representation. Such transformation and copying are platform-dependent, and are carried out by a set of functions within the native implementation (for our experiments, in a JNI module). This function set can either be user- or system-defined. In the latter case, the application programmer has to define an interface describing the data fields that need to be transformed and copied when morphing is performed. This interface has to be written in both native (in our case C) and agent (Java) code. System-provided state transformation functions rely on these interface definitions; they also rely on object reflection techniques to achieve transformation and copying.

An agent may also morph back from its native form to its neutral form, which happens when the agent decides to migrate. In this case, the agent first transforms and copies its native states into neutral states, then switches back into agent mode, and finally migrates, with the help from the underlying agent system[38]. In general, morphing may be triggered at any point during agent execution in response to externally generated events or by the agent itself in response to internal state changes. However, after morphing, an agent has to restart from a fixed entry point, which essentially requires an agent to record its (application) state prior to morphing.

The morphable agents designed and used with the ISDA and PSSPS applications described in this paper utilize two key abstractions: (1) invocation *adaptors* and (2) *events*.

The purpose of the adaptor is like that of the *policies* associated with objects described in [9, 16, 35]: it intercepts all incoming invocations to the morphable agent, 'translates' them to the form appropriate for the agent's current representation (neutral or native), and then directs the invocations to this representation's implementation. Each agent uses a native form adaptor and a neutral form adaptor at the same time, so that invocation clients of the agent can invoke the agent regardless of their current states. The system we are constructing assumes that each agent is morphed in its entirety, either residing in its native or its neutral state; this eliminates problems with partial state translation and state consistency when state is accessed simultaneously by native and neutral method implementations.

The purpose of events is to provide a uniform manner in which morphing is initiated, in response to the receipt of events that are internal or external to the agent. Internal events may be raised when certain state changes occur; external events may be raised by other agents or by a resource management system that has global knowledge of the agent program's behavior.

Application of Morphing to Sample Applications. Sample morphable agents have been constructed with Java, where specialized (morphed) versions of these agents are also available as native code for SUN Sparc/Solaris machines. The performance benefits of agent morphing have already been presented in tables 1 and 2, when applied to the Isosurface and Azimuth transform agents. In these particular examples, morphing overheads only come from native library loading and minimal application state translating and copying and are thus almost negligible. However, we expect such overheads to be higher when a code repository server is involved and/or when the amount of shared data is significant and morphing is applied more frequently. These agent realizations utilize the adaptors described in this section, using internally generated events. We manually program adaptors in our sample applications, but we believe that such adaptors can be readily generated by a compiler from IDL files.

The conditions under which morphing is applied are straightforward. In each example, when the amount of data being processed by agents increase (e.g., the ISDA application's visualization wishes to view more data or the PSSPS application's image resolution is increased), then the agent implementations

of these transformers do not deliver suitable performance. This fact is detected by inspection of internal data buffer fill levels[1]. To speed up data processing, the agent first acquires its native representation, then invokes an application-provided state-checkpointing method (if such a method was defined by the user), then transforms its state using the function set provided either by the system or the application programmer, and finally, initiates execution of its native form from a fixed entry point.

Discussion. Advantages of morphing include the performance improvements demonstrated in this section and also potential improvements concerning the predictability of agent execution. Predictability is particularly important for real-time and embedded applications and because Java code execution times are believed to be difficult to predict due to interpretation and garbage collection[25]. Difficulties with morphing arise from two sources. First, if native implementations cannot be acquired from a machine's local file system, then morphing overheads may become large due to the costs of access to remote repositories. Second, increased generality and complexity of native code compared to the sample agents used in our work may make morphing infeasible and/or require the provision of additional mechanisms to enable morphing. For example, with the sample applications and with the object realizations used in our research, agent safety may be guaranteed due to the sample objects' relative lack of internal complexity (e.g., no object-initiated file accesses) and due to the object system implementation's lightweight nature and heavy use of libraries. For general CORBA- or DCOM-based object implementations, guaranteeing improved performance or predictability as well as safety will require additional effort. Furthermore, internal native states like 'open file descriptors' cause problems for morphing not easily addressed for future agent systems (see [5] for a more detailed discussion of this topic).

3.2 Agent Fusion

Fusion Concepts. Fusion is useful for distributed agents that communicate within and across different machines. For closely coupled cooperating agents, communication overheads can constitute significant portions of their operating costs. For example, in the sample applications described above, when an 'upstream' agent filters out much of the data, then the remaining data handed off to the 'downstream' agent may not result in significant communication-based overheads. This is the case for the 'statistics' agent operating on the atmospheric data in the ISDA application, for example. Conversely, when such filtering does not remove much data, communication overheads may be substantial, as evident for many of the PSSPS application's agents.

The intent of agent fusion is to remove communication overheads from collaborating sets of agents and to enable optimizations across agent boundaries. Briefly, these overheads include actual network transport and protocol processing times, data copying costs, and thread/process scheduling and context switching costs that arise when tasks are performed by multiple vs. single agents residing on the same machine. Possible optimizations across agent boundaries are similar to those performed by compilers across procedure boundaries, including inter-procedural analysis leading to code or data motion and procedure integration where multiple overlapping procedures are combined into a smaller number of more efficient, combined procedures[3, 31, 32]. One particularly unfortunate communication cost is that incurred by multiple cooperating agents residing on the same machine, where their shared location is due to unforeseen agent migration actions (e.g., both agents 'found' interesting data on the same machine, or both agents moved to that machine due to local resource availability). In this case, it is clear that such agents would operate much more efficiently if they were placed into the same address space and used shared user-level threads, as this would reduce agent invocation costs to the costs of a few procedure calls (via adaptors). It would also eliminate overheads associated with the implementation of asynchronous invocations, such as the use of additional threads and their scheduling and synchronization, and additional data copying due to asynchrony and kernel/user space crossings.

In summary, while agent morphing specializes individual mobile agents, fusion performs runtime optimizations across multiple, cooperating agents. Fusion may be applied repeatedly, to create single efficient agents from multiple collaborating agents.

Application of Fusion to Sample Applications. Agent fusion is applied to those components in the

[1] More sophisticated techniques for first detecting and diagnosing performance problems with pipeline-structured applications are described in [22, 34]

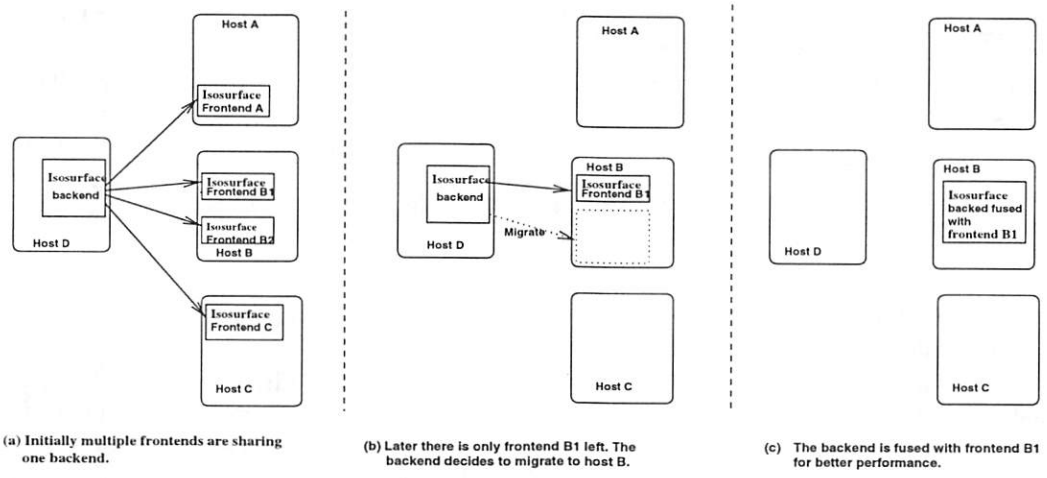


Figure 3: A scenario for fusion of Isosurface calculation front-end and back-end.

sample applications for which co-location on the same machine and with the same internal form (native or neutral) is likely to occur. For instance, Isosurface calculations in the ISDA application may be performed outside the scope of the visualization engine (if the visualization runs on a remote or weakly connected machine) and jointly with visualization. In fact, earlier versions of ISDA always performed Isosurface calculation within the visualization agent itself, as the visualization was running on a strongly connected and high end visualization engine (i.e., an SGI Octane). The versions of ISDA now used by end users do not wish to use such an instance of the visualization agent due to their desire to operate across a wider spectrum of machines and service a larger number of end users.

Figure 3 depicts the situation in which fusion is carried out on the Isosurface front-end and back-end, where the back-end does the actual isosurface calculation and then sends the computed isosurfaces as data to front-end on which visualization agent runs. Originally, there are multiple Isosurface front-ends, and the back-end is not co-located with any of the front-ends, with the intent of minimizing overall communication cost and avoiding burden the visualization engines with isosurface calculations. When all front-ends except one complete their execution, then the back-end might migrate to the remaining front-end's location, in order to reduce the communication cost between the two parties. The system is then able to fuse the two agents to further enhance performance. Table 4 shows the potential performance improvements resulting from such a fusion action. The same table also lists the results of fusing two agents in the PSSPS application: fir filtering and range processing.

In both of the experiments shown below, fusion is done manually through simulating the actions would have been taken by the fusion compiler. Such actions include procedure in-lining, data sharing, and asynchrony elimination.

Discussion of Results. The results in Table 4 indicate the benefits of agent fusion clearly. The table's first three columns show program performance when agents interact via remote object invocation, where references to other agents are remote object references received from a registry service. The fourth column shows the performance of the same programs in which references to cooperating agents are replaced by references to local objects, but object fusion has not been performed (ie., the compiler did not compile both agents as one unit). The next column shows performance subsequent to joint agent compilation.

These experiments demonstrate that it is highly desirable to put two agents into the same virtual machine and treat them as local object to each other if they are running on the same host and closely cooperating. In these experiments, the largest gain from agent fusion is due to the replacement of remote object invocation. The gains indicated in the last column are due solely to the compiler's use of global optimizations when multiple agents' code is combined;

Agents	Realization	Different Hosts	Same Host Diff Proc	Same process Remote Object	Local Object	Fully Fused	Fusion Benefit
Isosurface back & front-end	Java code	8,909	22,377	12,713	2,194	2,021	7.88%
	Compiled code	4,135	4,134	575	575	511	11.13%
Fir-Range processing	Java code	17,950	28,787	21,624	2,714	2,427	10.57%
	Compiled code	4,177	4,166	1,488	1,487	1,039	30.13%

Table 4: Result of fusion applied to different agent forms.

They range from 7.88% to 30.13% and are due to inter-class optimizations like inter-procedural aliasing and procedure in-lining, and most importantly, the elimination of asynchrony.

3.3 Summary

Experiments with sample applications built on top of our current infrastructure demonstrate that both morphing and fusion have significant performance benefits. Morphing provides 70% to 300% gains in performance, while fusion provides from 7.88% to 30.13% additional improvements after co-locating two agents in the same process.

The overheads caused by the acquisition of native realizations and by state transformation in morphing are relatively low, provided that morphing is not frequently invoked and that each run of the application lasts reasonablely long. We believe that the frequency of morphing will be low in most cases, as it needs to be invoked only once, unless migration is involved. However, migration itself tends to be an expensive activity; morphing will simply add some costs to this process. Similar arguments hold for object fusion.

4 Toward a System for Mobile Agent Optimization

The performance benefits derived from agent morphing and fusion presented in Sections 2 and 3 are significant. They are motivating us to construct an agent system within which agent/object–*objent*–programs are easily constructed and adapted at runtime. Such a system consists of contracts for performance requirement specifications, a notification system for contract monitoring, policies for application specified adaptation enactment, and finally, system or application defined adaptations.

This paper only addresses adaptation methods and adaptation enactment mechanisms unique to mobile agent-based applications. In our ongoing research, we are identifying other aspects unique to mobile agents and therefore, appropriately addressed by an adaptive Objent system. We expect to base this work on previous and current research in distributed adaptive systems[34, 6] and in distributed object systems[45].

Specifically, our Objent system has to address the following issues to support morphing and fusion adaptation:

1. Where and how are native agent forms created and maintained?
2. How does the system ensure consistency between the migratory and native versions of agent state?
3. When an agent's form is being or has been changed, can external agents not aware of this change continue invoking it?
4. What are useful fusion algorithms?

The basic fundamental components of the 'Objent' system we are developing have been described in a previous publication[2]. We next outline our solution approaches to the specific questions posed above.

Acquiring an Agent's Native Version. Agents are created and migrated in their platform-independent(neutral) forms. Their native forms may be created by (1) acquisition of a trusted native version from the agent's current execution site, involving agent retrieval from a local repository and/or its generation by a locally resident compiler, or (2) agent acquisition from a remote, trusted

repository to which providers submit agents in forms suitable for various platforms. Initial design ideas on such a repository are described in [2].

Consistency Between Multiple Agent Forms. An agent capable of morphing never has more than at most two implementations, a platform-neutral and a native one. At any one time, only one of these implementations is active. This implies that agent morphing necessitates state copying from the previously active to the new agent form. We intend to develop methods for full state copying, for partial state copying, and for permitting developers to provide specialized state maintenance methods.

Agent Invocation. When an agent changes its form, it is not likely that the agent's new form is able to efficiently interpret the objects passed to it via invocations to its old form. Moreover, one of the purposes of agent morphing is to enable efficient and direct communications between agents in their native forms whenever possible.

The solution being developed in our current research is one that permits the (inefficient) invocation of remote agents that differ in form, coupled with the implementation of notification protocols among agents that enable agents to switch to an efficient invocation protocol whenever possible. Specifically, we employ *adaptors* that are present in all agents capable of morphing. Each adaptor has two forms: (1) the neutral form is visible to the agent's neutral implementation; (2) the native form is visible to the native code. An adaptor is much like an object 'policy' in that all invocations in the respective agent forms are directed to the appropriate adaptor. The adaptor, then, knows about the agent's current form, has methods generated from its interface definition for request translation from one form to the other, and is able to deal with issues arising from the agent's concurrent invocation in both of its forms.

The overheads of using agent adaptors are small when agents communicate in the same form, as was the case for the overheads incurred by policies evaluated in [35]. When adaptors must translate between forms, overheads depend on the complexities of invocation parameters.

Fusion Algorithms. The fusion algorithms used in our current work carry out inter-procedural optimization, and they reduce or eliminate the multithreading overheads caused by asynchronous remote agent invocation. For example, in PSSPS, an asynchronous invocation is implemented as follows: with each method in an agent's interface definition, we associate a special modifier that denotes whether the method should be invoked synchronously (SYNC_IF_FUSED) or asynchronously (ASYNC_IF_FUSED) by fellow fused agent(s). An invocation to SYNC_IF_FUSED methods by a fellow fused agent(s) is replaced by a direct local procedure call. The fusion algorithm then applies interprocedural analysis to perform aliasing and, in the case of SYNC_IF_FUSED methods, procedure inlining. Aliasing attempts to eliminate unnecessary data copying, since data formerly located in different address spaces or on different hosts may potentially be shared subsequent to agent fusion and colocation.

Fusion may be applied repeatedly, possibly later followed by agent 'splitting', if indicated. Agent 'splitting' is an agent adaptation method we are aware of, it applies *program slicing* to an agent operating on a distributed data set and distributes agent slices so that each agent slice operates on some local data which is a subset of the distributed data set.

5 Related Work

Recent active research on mobile agent systems concerns the areas of agent facility standardization, mobile agent system interoperability, and operating system support [17, 20, 24, 38]. The platforms developed by such works provide the basic agent system functionality upon which our runtime system is built. We add to this functionality the ability to adapt mobile agent and we add the event mechanisms necessary for building dynamic runtime support for monitoring and for adaptation initiation and enactment.

Research results from software specialization systems like SPIN, Exokernel, and Synthetix may be applied to our adaptable agent architecture to customize the agent system itself and/or individual agents. We will focus on customization issues more specific to mobile agent environments.

We have already benefited from research on object policies and on meta-objects[35, 16] to develop the 'adaptor' concept presented in section 3. Our work will also take advantage of current research on quality of service infrastructures like BBN's QuO[45] and Honeywell's ARA[34], but we will adapt their techniques to the mobile agent domain targeted by our work.

Our work is part of a broader project described in [1], with early results are presented in [2].

6 Conclusions and Future Work

Agent computing is subject to several inefficiencies, some of which are due to the complexities of the environments in which mobile agents are deployed. Our research is exploring runtime adaptation and agent specialization to improve the performance of agent-based programs, aiming at enabling programmers to employ these techniques and runtime adaptation in general, to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. We explore the effects of using two specialization approaches, morphing and fusion, on a single mobile agent and on several cooperating agents. Our experimental results with two sample applications, ISDA and PSSPS, show that such specialization approaches result in considerable performance improvement.

We have built a preliminary infrastructure for online morphing, which offers mechanism for interlanguage remote invocation using the model of invocation adaptors developed in our research. Infrastructures and mechanisms are applied to the ISDA and PSSPS distributed high performance applications. Also used with these applications is a realization of mobile event channels that allow reliable event delivery during end-point migration.

Our future work concerns systematic support for specialization approaches like morphing, fusion and others such as slicing. This support will comprise event mechanisms and quality of service infrastructure, both of which are important to a general agent adaptation system. We will also work on compilers for agent fusion and the adaptation of agent invocations.

Acknowledgments. Prof. Raja Das is participating in the Objent system design, with focus on the application of compilation methods to improve agent or object performance, including the use of fusion techniques. Prof. Mustaque Ahamad has been investigating the overheads of agent communications (e.g., Java RMI) and methods for creating quality-controlled Java-based objects. Greg Eisenhauer and Beth Plale are responsible for the ISDA application, Davis King provided the code for Isosurface calculation, and Fabian Bustamante is currently creating additional elements part of the ISDA application. Rajkumar Krishnamurthy provided the SAR benchmark code used in our work.

References

[1] Mustaque Ahamad, Raja Das, and Karsten Schwan. Integrating object and agent technologies for high-end collaborative applications. http://www.cc.gatech.edu/systems/facstaff/ahamad/object.html.

[2] Mustaque Ahamad, Raja Das, Karsten Schwan, Sumeer Bhola, Fabian Bustamante, Greg Eisenhauer, Jeremy Heiner, Vijaykumar Krishnaswamy, Todd Rose, Beth Schroeder, and Dong Zhou. Agent and object technologies for high-end collaborative applications. In *OOPSLA'97 Workshop on Java-Based Paradigms for Mobile Agent Facilities*, 1997.

[3] Michael Burke and Jong-Deok Choi. Precise and efficient integration of interprocedural alias information into data-flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1), Mar. 1992.

[4] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. IBM T.J. Watson Research Center, 1995.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[6] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Aug. 1998.

[7] COBS: Configurable OBjectS for High Performance Systems. College of computing, georgia institute of technology. http://www.cc.gatech.edu/systems/projects/COBS/.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] Ahmed Gheith and Karsten Schwan. Chaos-arc - kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[10] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification (Java Series)*. Addison-Wesley, 1996.

[11] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[12] Robert Grossman. The terabyte challenge: An open, distributed testbed for managing and mining massive data sets. http://www.lac.uic.edu/hpcc-grossman.html.

[13] Habanero. National Center for Supercomputing Applications and university of illinois at urbana-champaign. http://notme.ncsa.uiuc.edu/SDG/Software/Habanero.

[14] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T.J. Watson Research Center, 1995.

[15] Delbert Hart and Eileen Kraemer. An agent-based perspective on distributed monitoring and steering. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Welches, Oregon, Aug. 1998.

[16] Jun ichiro Itoh, Rodger Lea, and Yasuhiko Yokote. Using meta-objects to support optimisation in the apertos operating system. In *Proceedings of USENIX Conference on Object-Oriented Technologies (COOTS)*, Jun. 1995.

[17] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, May 1995.

[18] Keith D. Kotay and David Kotz. Transportable agents. In *CIKM Workshop on Intelligent Information Agents*, Gaithersburg, Maryland, Dec. 1994.

[19] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementation of java remote method invocation (rmi). In *Proceedings of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[20] Danny B. Lange and Daniel T. Chang. White paper. IBM Aglets Workbench, Sep. 1996.

[21] C. Lee, C. Kesselman, and S. Schwab. Near-real-time satellite image processing: Metacomputing in cc++. *Computer Graphics and Applications*, 16(4), 1996.

[22] Vernard Martin and Karsten Schwan. ILI: An adaptive infrastructure for dynamic interactive distributed systems. In *4th International Conference on Configurable Distributed Systems*. IEEE, 1998.

[23] Sun Microsystems. Java on solaris 2.6: A white paper. http://wwwwseast2.usec.sun.com/solaris/java/wp-java/.

[24] Dejan S. Milojicicc, William LaForge, and Deepika Chauhan. Mobile objects and agents (moa). In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[25] Akihiko Miyoshi and Takuro Kitayama. Implementation and evaluation of real-time java threads. In *Proceedings of Real-Time Systems Symposium*, Dec. 1997.

[26] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[27] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2), 1998.

[28] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, 1995.

[29] William Ribarsky, Yves Jean, Thomas Kindler, Weiming Gu, Gregory Eisenhauer, Karsten Schwan, , and Fred Alyea. An integrated approach for steering, visualization, and analysis of atmospheric simulations. In *Proceedings IEEE Visualization '95*, 1995.

[30] William Ribarsky, Yves Jean, Song Zou, Karsten Schwan, Bobby Sumner, , and Onome Okuma. A heterogeneous environment for visual steering of computer simulations. Submitted to IEEE Computer Graphics & Applications.

[31] Stephen Richarson and mahadevan Ganapathi. Code optimization across procedures. *IEEE Computer*, Feb. 1989.

[32] Stephen Richarson and mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software–Practice and Experience*, 19(2), Feb. 1989.

[33] Daniela Rosu, Karsten Schwan, and Sudhakar Yalamanchili. Fara - a framework for adaptive resource allocation in complex real-time systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Denver, USA, Jun. 1998.

[34] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, USA, Dec. 1997.

[35] D. Silva and K. Schwan. Ctk: Configurable object abstractions for multiprocessors. Technical Report GIT-CC-97-03, College of Computing, Georgia Institute of Technology, 1997.

[36] D. Silva, K. Schwan, and G. Eisenhauer. Configurable distributed retrieval of scientific data. In *Second International Conference on Configurable Distributed Systems (CDS'98)*, Maryland, May 1998.

[37] Sandeep K. Singhal, Binh Q. Nguyen, Richard Redpath, Michael Fraenkel, and Jimmy Nguyen. Building high-performance applications and servers in java. In *ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications*, Atlanta, Georgia, October 1997.

[38] Markus Straber, Joachim Baumann, and Fritz Hohl. Mole-a java based mobile agent system. In *ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[39] J.S. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. IPPS 97*, 1997.

[40] VizAD. Space science and engineering center university of wisconsin - madison. http://www.ssec.wisc.edu/ billh/visad.html.

[41] Glen H. Wheless, Cathy M. Lascara, Donald P. Brutzman, William Sherman, William L. Hibbard, and

Brian E. Paul. Chesapeake bay: Interacting with a physical/biological model. *IEEE Computer Graphics and Applications*, 16(4), July/August 1996.

[42] J. White. Mobile agents. Telescript Technical Whitepaper, General Magic, Inc., oct. 1995.

[43] et. al. Wollrath. A distributed object model for the java system. *Computing Systems*, 9(4):265–290, 1996.

[44] Rich Wolski. Dynamically forecasting network performance to support dynamic scheduling using the Network Weather Service. In *Proceedings of 6th High-Performance Distributed Computing (HPDC6)*. IEEE, 1997.

[45] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, January 1997.

[46] B. Zuerndorfer and G. A. Shaw. Sar processing for rassp application. In *Proceedings of 1st Annual RASSP Conference*, Arlington, VA., Aug. 1994.

Applying Optimization Principle Patterns to Design Real-time ORBs

Irfan Pyarali, Carlos O'Ryan, Douglas Schmidt,
Nanbor Wang, and Vishal Kachroo
{irfan,coryan,schmidt,vishal,nanbor}@cs.wustl.edu
Washington University
Campus Box 1045
St. Louis, MO 63130†

Aniruddha Gokhale*
gokhale@research.bell-labs.com
Bell Labs, Lucent Technologies
600 Mountain Ave Rm 2A-442
Murray Hill, NJ 07974

Abstract

First-generation CORBA middleware was reasonably successful at meeting the demands of request/response applications with best-effort quality of service (QoS) requirements. Supporting applications with more stringent QoS requirements poses new challenges for next-generation real-time CORBA middleware, however. This paper provides three contributions to the design and optimization of real-time CORBA middleware. First, we outline the challenges faced by real-time ORBs implementers, focusing on optimization principle patterns that can be applied to CORBA's Object Adapter and ORB Core. Second, we describe how TAO, our real-time CORBA implementation, addresses these challenges and applies key ORB optimization principle patterns. Third, we present the results of empirical benchmarks that compare the impact of TAO's design strategies on ORB efficiency, predictability, and scalability.

Our findings indicate that ORBs must be highly configurable and adaptable to meet the QoS requirements for a wide range of real-time applications. In addition, we show how TAO can be configured to perform predictably and scalably, which is essential to support real-time applications. A key result of our work is to demonstrate that the ability of CORBA ORBs to support real-time systems is mostly an implementation detail. Thus, relatively few changes are required to the standard CORBA reference model and programming API to support real-time applications.

1 Introduction

Many companies and research groups are developing distributed applications using middleware components like CORBA Object Request Brokers (ORBs) [1]. CORBA helps to improve the flexibility, extensibility, maintainability, and reusability of distributed applications [2]. However, a growing class of distributed real-time applications also require ORB middleware that provides stringent quality of service (QoS) support, such as end-to-end priority preservation, hard upper bounds on latency and jitter, and bandwidth guarantees [3]. Figure 1 depicts the layers and components of an ORB endsystem that must be carefully designed and systematically optimized to support end-to-end application QoS requirements.

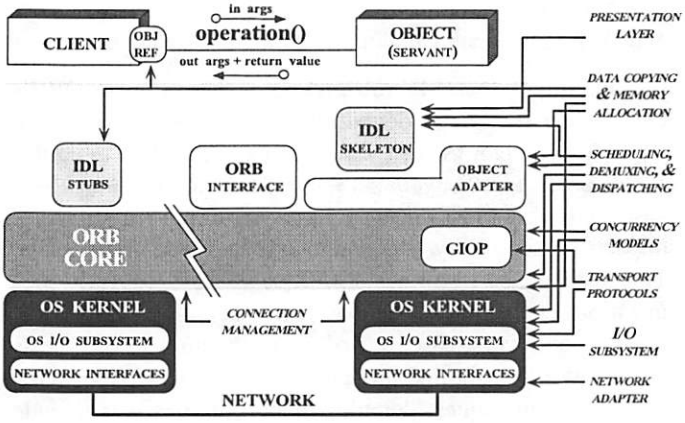


Figure 1: Real-time Features and Optimizations Necessary to Meet End-to-end QoS Requirements in ORB Endsystems

First-generation ORBs lacked many of the features and optimizations [4, 5, 6, 7] shown in Figure 1. This situation was not surprising, of course, since the focus at that time was largely on developing core infrastructure components, such as the ORB and its basic services, defined by the OMG specifications [8]. In contrast, second-generation ORBs, such as The ACE ORB (TAO) [9], explicitly focus on providing end-to-end QoS guarantees to applications *vertically* (*i.e.*, network

*Work done by the author while at Washington University.

†This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, Motorola, Siemens ZT, and Sprint.

interface ↔ application layer) and *horizontally* (*i.e.*, end-to-end) integrating highly optimized CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces.

Our previous research has examined many dimensions of high-performance and real-time ORB endsystem design, including static [9] and dynamic [10] scheduling, event processing [11], I/O subsystem integration [12], ORB Core connection and concurrency architectures [7], systematic benchmarking of multiple ORBs [4], and design patterns for ORB extensibility [13]. This paper focuses on four more dimensions in the high-performance and real-time ORB endsystem design space: *Object Adapter and ORB Core optimizations for (1) request demultiplexing, (2) collocation, (3) memory management, and (4) ORB protocol overhead.*

The optimizations used in TAO are guided by a set of *principle patterns* [14] that have been applied to optimize middleware [15] and lower-level networking software [16], such as TCP/IP. Optimization principle patterns document rules for avoiding common design and implementation problems that degrade the performance, scalability, and predictability of complex systems. The optimization principle patterns we applied to TAO include: *optimizing for the common case; eliminating gratuitous waste; shifting computation in time such as precomputing; avoiding unnecessary generality; passing hints between layers; not being tied to reference implementations; using specialized routines; leveraging system components by exploiting locality; adding state; and using efficient data structures.* Below, we outline how these optimization principle patterns address the following TAO Object Adapter and ORB Core design and implementation challenges.

Optimizing request demultiplexing: The time an ORB's Object Adapter spends demultiplexing requests to target object implementations, *i.e.*, servants, can constitute a significant source of ORB overhead for real-time applications. Section 2 describes how Object Adapter demultiplexing strategies impact the scalability and predictability of real-time ORBs. This section also illustrates how TAO's Object Adapter optimizations enable constant time request demultiplexing in the average- and worst-case, regardless of the number of objects or operations configured into an ORB. The principle patterns that guide our request demultiplexing optimizations include *precomputing*, *using specialized routines*, *passing hints in protocol headers*, and *not being tied to reference models*.

Optimizing collocation: The principle pattern of relaxing system requirements enables TAO to minimize the run-time overhead for *collocated* objects, *i.e.*, objects that reside in the same address space as their client(s). Operations on collocated objects are invoked on servants directly in the context of the calling thread, thereby transforming operation invocations into local virtual method calls. Section 3.1 describes how TAO's collocation optimizations are completely transparent to clients, *i.e.*, collocated objects can be used as regular CORBA objects, with TAO handling all aspects of collocation.

Optimizing memory management: ORBs allocate buffers to send and receive (de)marshaled data. It is important to optimize these allocations since they are a significant source of dynamic memory management and locking overhead. Section 3.2 describes the mechanisms used in TAO to allocate and manipulate the internal buffers it uses for parameter (de)marshaling. We illustrate how TAO minimizes fragmentation, data copying, and locking for most application use-cases. The principle patterns of *exploiting locality* and *optimizing for the common case* influence these optimizations.

Minimizing ORB protocol overhead: Real-time systems have traditionally been developed using proprietary protocols that are hard-coded for each application or application family. In theory, the standard CORBA GIOP/IIOP protocols obviate the need for proprietary protocols. In practice, however, many developers of real-time applications are justifiably concerned that standard CORBA protocols incur excessive overhead. Section 3.3 shows how TAO can be configured to reduce the overhead of GIOP/IIOP without affecting the standard CORBA programming APIs exposed to application developers. This optimization is based on the principle pattern of *avoiding unnecessary generality*.

The remainder of this paper is organized as follows: Section 2 outlines the Portable Object Adapter (POA) architecture of CORBA ORBs and evaluates the design and performance of POA optimizations used in TAO; Section 3 outlines the ORB Core architecture of CORBA ORBs and evaluates the design and performance of ORB Core optimizations used in TAO; Section 4 describes related work; and Section 5 provides concluding remarks.

2 Optimizing the POA for Real-time Applications

2.1 POA Overview

The OMG CORBA 2.2 specification [1] standardizes several components on the server-side of CORBA-compliant ORBs. These components include the Portable Object Adapter (POA), standard interfaces for object implementations (*i.e.*, servants), and refined definitions of skeleton classes for various programming languages, such as Java and C++ [2].

These standard POA features allow application developers to write more flexible and portable CORBA servers [17]. They also make it possible to conserve resources by activating objects on-demand [18] and to generate "persistent" object references [19] that remain valid after the originating server pro-

cess terminates. Server applications can configure these new features portably using *policies* associated with each POA.

CORBA 2.2 allows server developers to create *multiple* Object Adapters, each with its own set of policies. Although this is a powerful and flexible programming model, it can incur significant run-time overhead because it complicates the request demultiplexing path within a server ORB. This is particularly problematic for real-time applications since naive Object Adapter implementations can increase priority inversion and non-determinism [6].

Optimizing a POA to support real-time applications requires the resolution of several design challenges. This section outlines these challenges and describes the optimization principle patterns we applied to maximize the predictability, performance, and scalability of TAO's POA. These POA optimizations include constant-time demultiplexing strategies, reducing run-time object key processing overhead during upcalls, and generally optimizing POA predictability and reducing memory footprint by selectively omitting non-deterministic POA features.

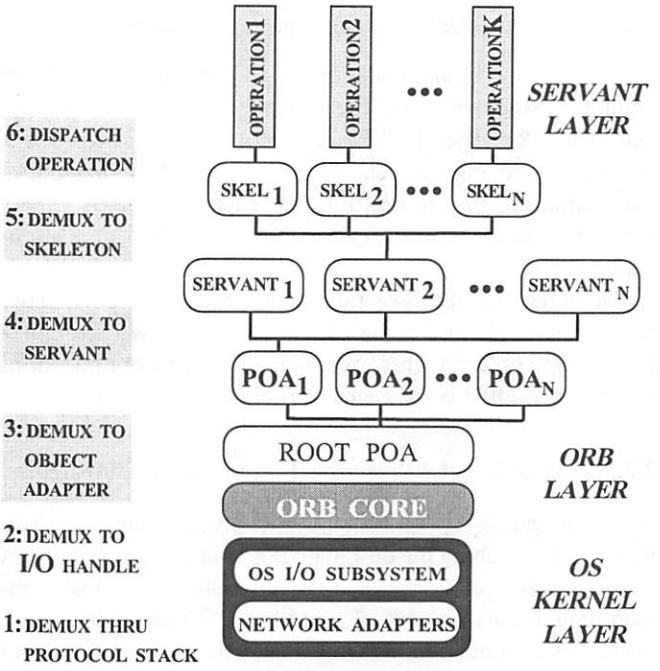


Figure 2: CORBA 2.2 Logical Server Architecture

2.2 Optimizing POA Demultiplexing

Scalable and predictable POA demultiplexing is important for many applications such as real-time stock quote systems [20] that service a large number of clients, and avionics mission systems [11] that have stringent hard real-time timing constraints. Below, we outline the steps involved in demultiplexing a client request through the server-side of a CORBA ORB and then qualitatively and quantitatively evaluate alternative demultiplexing strategies.

2.2.1 Overview of CORBA Request Demultiplexing

A standard GIOP-compliant client request contains the identity of its object and operation. An object is identified by an object key, which is an `octet sequence`. An operation is represented as a `string`. As shown in Figure 2, the ORB endsystem must perform the following demultiplexing tasks:

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, starting from the network interface, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket layer), where the data is passed to the ORB Core in a server process.

Steps 3, and 4: The ORB Core uses the addressing information in the client's object key to locate the appropriate POA and servant. POAs can be organized hierarchically. Therefore, locating the POA that contains the designated servant can involve a number of demultiplexing steps through the nested POA hierarchy.

Step 5 and 6: The POA uses the operation name to find the appropriate IDL skeleton, which demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers to implement the object's operation.

The conventional deeply-layered ORB endsystem demultiplexing implementation shown in Figure 2 is generally inappropriate for high-performance and real-time applications for the following reasons [21]:

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched as incoming client requests ascend through the processing layers in an ORB endsystem. Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an Object Adapter.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level quality of service (QoS) information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, an Object Adapter may demultiplex packets according to their FIFO order of arrival. FIFO demultiplexing can cause higher priority packets to wait for a non-deterministic period of time while

lower priority packets are demultiplexed and dispatched [12].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [4, 6] show that conventional ORBs spend ~17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform, scalable QoS guarantees to real-time applications.

The remainder of this section focuses on demultiplexing optimizations performed at the ORB layer, *i.e.*, steps 3 through 6. Information on OS kernel layer demultiplexing optimizations for real-time ORB endsystems is available in [22, 12].

2.2.2 Overview of Alternative Demultiplexing Strategies

As illustrated in Figure 2, demultiplexing a request to a servant and dispatching the designated servant operation involves several steps. Below, we qualitatively outline the most common demultiplexing strategies used in CORBA ORBs. Section 2.2.3 then quantitatively evaluates the strategies that are appropriate for each layer in the ORB.

Linear search: This strategy searches through a table sequentially. If the number of elements in the table is small, or the application has no stringent QoS requirements, linear search may be an acceptable demultiplexing strategy. For real-time applications, however, linear search is undesirable since it does not scale up efficiently or predictably to a large number of servants or operations. In this paper, we evaluate linear search only to provide an upper-bound on worst-case performance, though some ORBs [4] use linear search for operation demultiplexing.

Binary search: Binary search is a more scalable demultiplexing strategy than linear search since its $O(\lg n)$ lookup time is effectively constant for most applications. However, insertions and deletions can be complicated since data must be sorted for the binary search algorithm to work correctly. Therefore, binary search is particularly useful for ORB operation demultiplexing since all insertions and sorting can be performed off-line by an IDL compiler. In contrast, using binary search to demultiplex requests to servants is more problematic since servants can be inserted or removed dynamically at run-time.

Dynamic hashing: Many ORBs use dynamic hashing as their Object Adapter demultiplexing strategy. Dynamic hashing provides $O(1)$ performance for the average case and supports dynamic insertions more readily than binary search. However, due to the potential for collisions, its worst-case execution time is $O(n)$, which makes it inappropriate for hard real-time applications that require efficient and predictable worst-case ORB behavior. Moreover, depending on the hash algorithm, dynamic hashing often has a fairly high constant overhead [6].

Perfect hashing: If the set of operations or servants is known *a priori*, dynamic hashing can be improved by precomputing a collision-free *perfect hash function* [23]. Perfect Hashing is based on the principle pattern of *precomputing* and *using specialized routines*. A demultiplexing strategy based on perfect hashing executes in constant time and space. This property makes perfect hashing well-suited for deterministic real-time systems that can be configured statically [6], *i.e.*, the number of objects and operations can be determined off-line.

Active demultiplexing: Although the number and names of operations can be known *a priori* by an IDL compiler, the number and names of servants are generally more dynamic. In such cases, it is possible to use the object ID and POA ID stored in an object key to index directly into a table managed by an Object Adapter. Active demultiplexing uses the principle pattern of *relaxing system requirements*, *not being tied to reference models*, and *passing hints in headers*. This so-called *active demultiplexing* [6] strategy provides a low-overhead, $O(1)$ lookup technique that can be used throughout an Object Adapter.

Table 1 summaries the demultiplexing strategies considered in the implementation of TAO's POA.

Strategy	Search Time	Comments
Linear Search	$O(n)$	Simple to implement Does not scale
Binary Search	$O(\lg n)$	Additions/deletions are expensive
Dynamic Hashing	$O(1)$ average case $O(n)$ worst case	Hashing overhead
Perfect Hashing	$O(1)$ worst case	For static configurations, generate collision-free hashing functions
Active Demuxing	$O(1)$ worst case	For system generated keys, add direct indexing information to keys

Table 1: Summary of Alternate POA Demultiplexing Strategies

2.2.3 The Performance of Alternative POA Demultiplexing Strategies

Section 2.2.1 describes the demultiplexing steps a CORBA request goes through before it is dispatched to a user-supplied servant method. These demultiplexing steps include finding the Object Adapter, the servant, and the skeleton code. This section empirically evaluates the strategies that TAO uses for

each demultiplexing step. All POA demultiplexing measurements were conducted on an UltraSPARC-II with two 300 MHz CPUs, a 512 Mbyte RAM, running SunOS 5.5.1, and C++ Workshop Compilers version 4.2.

POA lookup: An ORB Core must locate the POA corresponding to an incoming client request. Figure 2 shows that POAs can be nested arbitrarily. Although nesting provides a useful way to organize policies and namespaces hierarchically, the POA's nesting semantics complicate demultiplexing compared with the original CORBA Basic Object Adapter (BOA) demultiplexing [6] specification.

We conducted an experiment to measure the effect of increasing the POA nesting level on the time required to lookup the appropriate POA in which the servant is registered. We used a range of POA depths, 1 through 25. The results are shown in Figure 3.

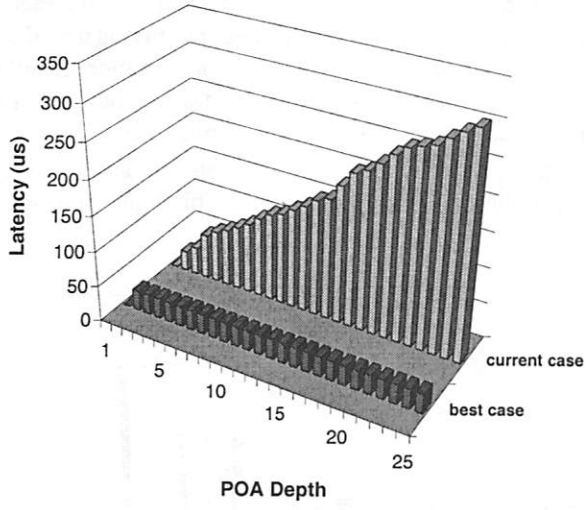


Figure 3: Effect of POA Depth on POA Demultiplexing Latency

Since most ORB server applications do not have deeply nested POA hierarchies, TAO currently uses a POA demultiplexing strategy where each POA finds its child using dynamic hashing and delegates to the child POA where this process is repeated until the search is complete. This POA demultiplexing strategy results in $O(n)$ growth for the lookup time and does not scale up to deeply nested POAs. Therefore, we are adding active demultiplexing to the POA lookup phase, which operates as follows:

1. All lookups start at the `RootPOA`.
2. The `RootPOA` will maintain a `POA table` that points to all the POAs in the hierarchy.
3. Object keys will include an index into the `POA table` to identify the POA where the object was activated. TAO's ORB Core will use this index as the active demultiplexing key.
4. In some cases, the POA name also may be needed, *e.g.*, if the POA is activated on-demand. Therefore, the object reference will contain both the name and the index.

Using active demultiplexing for POA lookup should provide optimal predictability and scalability, just as it does when used for servant demultiplexing, which is described next.

Servant demultiplexing: Once the ORB Core demultiplexes a client request to the right POA, this POA demultiplexes the request to the correct servant. The following discussion compares the various servant demultiplexing techniques described in Section 2.2.2. TAO uses the Service Configurator [24], Bridge, and Strategy design patterns [25] to defer the configuration of the desired servant demultiplexing strategy until ORB initialization, which can be performed either *statically* (*i.e.*, at compile-time) or *dynamically* (*i.e.*, at runtime) [13]. Figure 4 illustrates the class hierarchy of strategies that can be configured into TAO's POAs.

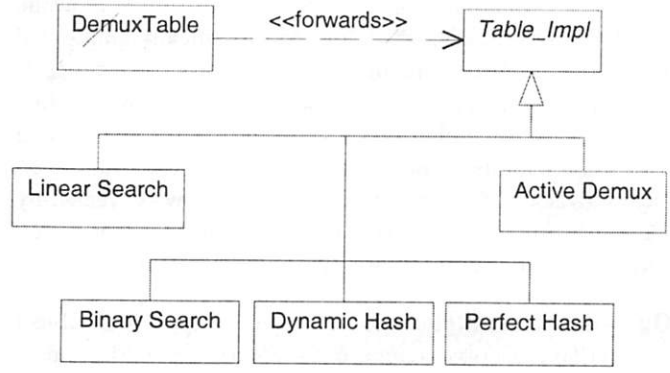


Figure 4: TAO's Class Hierarchy for POA Active Object Map Strategies

To evaluate the scalability of TAO, our experiments used a range of servants, 1 to 500 by increments of 100, in the server. Figure 5 shows the latency for servant demultiplexing as the number of servants increases. This figure illustrates that active demultiplexing is a highly predictable, low-latency servant lookup strategy. In contrast, dynamic hashing incurs higher constant overhead to compute the hash function. Moreover, its performance degrades gradually as the number of servants increases and the number of collisions in the hash table increase. Likewise, linear search does not scale for any realistic system, *i.e.*, its performance degrades rapidly as the number of servants increase.

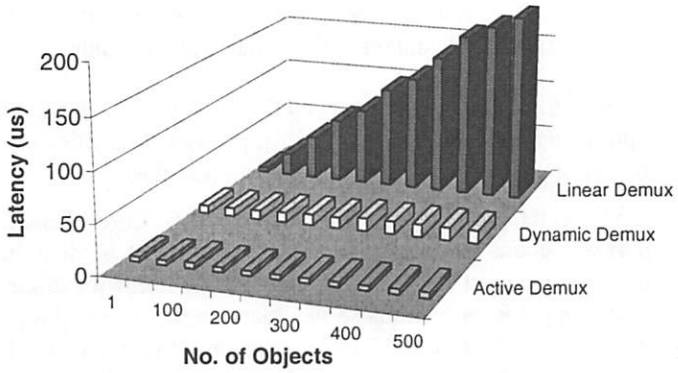


Figure 5: Servant Demultiplexing Latency with Alternative Search Techniques

Note that we did not implement the perfect hashing strategy for servant demultiplexing. Although it is possible to know the set of servants on each POA for certain statically configured applications *a priori*, creating perfect hash functions repeatedly during application development is tedious. We omitted binary search for similar reasons, *i.e.*, it requires maintaining a sorted active object map every time an object is activated or deactivated. Moreover, since the object key is created by a POA, active demultiplexing provides equivalent, or better, performance than perfect hashing or binary search.

Operation demultiplexing: The final step at the Object Adapter layer involves demultiplexing a request to the appropriate skeleton, which demarshals the request and dispatches the designated operation upcall in the servant. To measure operation demultiplexing overhead, our experiments defined a range of operations, 1 through 50, in the IDL interface.

For ORBs like TAO that target real-time embedded systems, operation demultiplexing must be efficient, scalable, and predictable. Therefore, we generate efficient operation lookup using GPERF [23], which is a freely available perfect hash function generator we developed.

GPERF [26] automatically constructs perfect hash functions from a user-supplied list of keywords. In addition to the perfect hash functions, GPERF can also generate linear and binary search strategies.

Figure 6 illustrates the interaction between the TAO IDL compiler and GPERF. When perfect hashing, linear search and binary search operation demultiplexing strategies are selected, TAO's IDL compiler invokes GPERF as a co-process to generate an optimized lookup strategy for operation names in IDL interfaces.

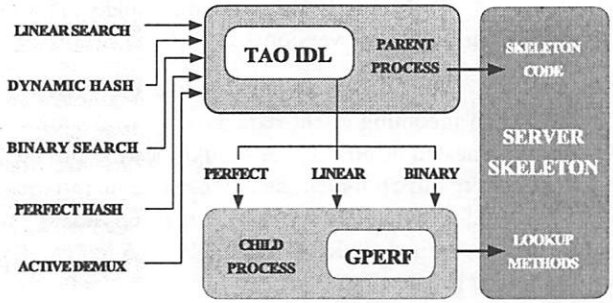


Figure 6: Integrating TAO's IDL Compiler and GPERF

The lookup key for this phase is the operation name, which is a `string` defined by developers in an IDL file. However, it is not permissible to modify the operation `string` name to include active demultiplexing information. Since active demultiplexing cannot be used without modifying the GIOP protocol.[1] TAO uses perfect hashing for operation demultiplexing. Perfect hashing is well-suited for this purpose since all operations names are known at compile time.

Figure 7 plots operation demultiplexing latency as a function of the number of operations. This figure illustrates that

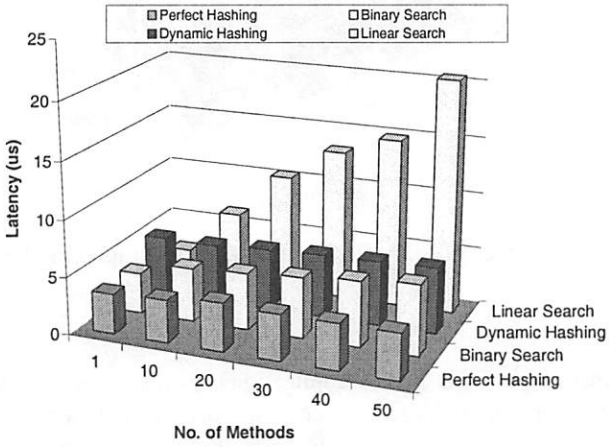


Figure 7: Operation Demultiplexing Latency with Alternative Search Techniques

perfect hashing is extremely predictable and efficient, outperforming dynamic hashing and binary search. As expected, linear search depends on the number and ordering of operations, which complicates worst-case schedulability analysis for real-time applications.

[1] We are investigating modifications to the GIOP protocol for hard real-time systems that possess stringent latency and message-footprint requirements.

Optimizing servant-based lookups: When a CORBA request is dispatched by the POA to the servant, the POA uses the Object Id in the request header to find the servant in its Active Object Map. Section 2.2.3 describes how TAO's lookup strategies provide efficient, predictable, and scalable mechanisms to dispatch requests to servants based on Object Ids. In particular, TAO's Active Demultiplexing strategy enables constant $O(1)$ lookup in the average- and worst-case, regardless of the number of servants in a POA's Active Object Map.

However, certain POA operations and policies require lookups on Active Object Map to be based on the *servant pointer* rather than the Object Id. For instance, the `_this` method on the servant can be used with the `IMPLICIT_ACTIVATION` POA policy outside the context of request invocation. This operation allows a servant to be activated implicitly if the servant is not already active. If the servant is already active, it will return the object reference corresponding to the servant.

Unfortunately, naive POA's Active Object Map implementations incur worst-case performance for servant-based lookups. Since the primary key is the Object Id, servant-based lookups degenerate into a linear search, even when Active Demultiplexing is used for the Object Id-based lookups. As shown in Figure 5, linear search is prohibitively expensive as the number of servants in the Active Object Map increases. This overhead is particularly problematic for real-time applications, such as avionics mission computing systems [11], that (1) create a large number of objects using `_this` during their initialization phase and (2) must reinitialize rapidly to recover from transient power failures.

To alleviate servant-based lookup bottlenecks, we apply the principle pattern of *adding extra state* to the POA in the form of a *Reverse-Lookup* map that associates each servant with its Object Id in $O(1)$ average-case time. In TAO, this Reverse-Lookup map is used in conjunction with the Active Demultiplexing map that associates each Object Id to its servant. Figure 8 shows the time required to find a servant, with and without the Reverse-Lookup map, as the number of servants in a POA increases.

Servants are allocated from arbitrary memory locations. Since we have no control over the pointer value format, TAO uses a hash map for the Reverse-Lookup map. The value of the servant pointer is used as the hash key. Although hash maps do not guarantee $O(1)$ worst-case behavior, they do provide a significant average-case performance improvement over linear search.

A Reverse-Lookup map can be used only with the `UNIQUE_ID` POA policy since with the `MULTIPLE_ID` POA policy, a servant may support many Object Ids. This constraint is not a shortcoming since servant-based lookups are only required with the `UNIQUE_ID` policy. One downside of adding a Reverse-Lookup map to the POA, however, is the increased

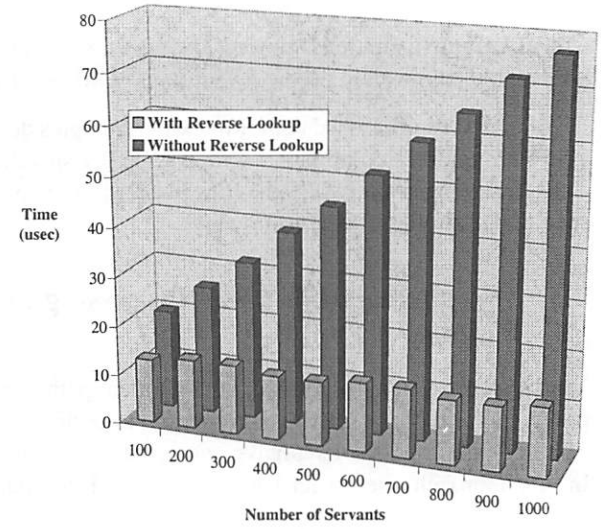


Figure 8: Benefits of Adding a Reverse-Lookup Map to the POA

overhead of maintaining an additional table in the POA. For every object activation and deactivation, two updates are required in the Active Object Map: (1) to the Reverse-Lookup map and the (2) to the Active Demultiplexing map used for Object Id-based lookups. However, this additional processing does not affect the critical path of Object Id-based lookups during run-time.

Summary of TAO's POA demultiplexing strategies: Based on the results of our benchmarks described above, Figure 9 summarizes the demultiplexing strategies that we have determined to be most appropriate for real-time applications [11]. Figure 9 shows the use of active demultiplex-

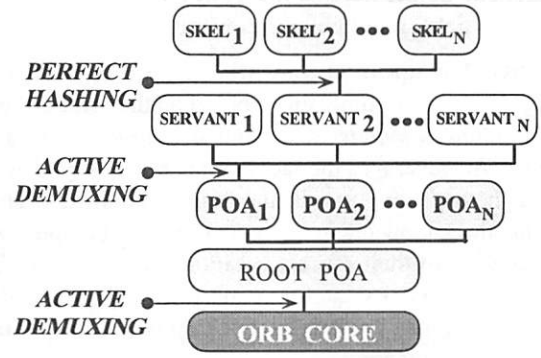


Figure 9: TAO's Default Demultiplexing Strategies

ing for the POA names, active demultiplexing for the servants, and perfect hashing for the operation names. Our previous experience [27, 4, 28, 6, 7] measuring the performance of

CORBA implementations showed TAO is more efficient and predictable than widely used conventional CORBA ORBs.

All of TAO's optimized demultiplexing strategies described above are entirely compliant with the CORBA specification. Thus, no changes are required to the standard POA interfaces specified in CORBA specification [1].

2.3 Optimizing Object Key Processing in POA Upcalls

Motivation: Since the POA is in the critical path of request processing in a server ORB, it is important to optimize its processing. Figure 10 shows a naive way to parse an object key. In this approach, the object key is parsed and the individual

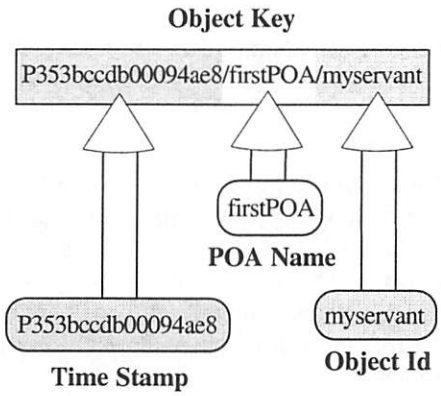


Figure 10: Naive Parsing of Object Keys

fields of the key are stored in separate components. Unfortunately, this approach (1) allocates memory dynamically for each individual object key field and (2) copies data to move the object key fields into individual objects.

TAO's object key upcall optimizations: TAO provides the following object key optimizations based on the principle patterns of *avoiding obvious waste* and *avoiding unnecessary generality*. TAO leverages the fact that the object key is available through the entire upcall and is not modified. Thus, the individual components in the object key can be optimized to point directly to their correct locations, as shown in Figure 11. This eliminates wasteful memory allocations and data copies. This optimization is entirely compliant with the standard CORBA specification.

2.4 Optimizing POA Predictability and Minimizing Footprint

Motivation: To adequately support real-time applications, an ORB's Object Adapter must be *predictable* and *minimal*.

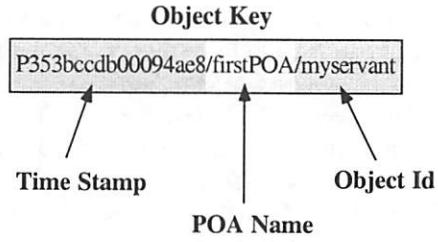


Figure 11: TAO's Optimized Parsing of Object Keys

For instance, it must omit non-deterministic operations to improve end-to-end predictability. Likewise, it must provide a minimal memory footprint to support embedded systems [15].

TAO's predictability optimizations: Based on the principle patterns of *avoiding unnecessary generality* and *relaxing system requirements*, we enhanced TAO's POA to selectively disable the following features in order to improve end-to-end predictability of request processing:

- **Servant Managers are not required:** There is no need to locate servants in a real-time environment since all servants must be registered with POAs *a priori*.

- **Adapter Activators are not required:** Real-time applications create all their POAs at the beginning of execution. Therefore, they need not use or provide an adapter activator. The alternative is to create POAs during request processing, in which case end-to-end predictability is hard to achieve.

- **POA Managers are not required:** The POA must not introduce extra levels of queueing in the ORB. Queueing can cause priority inversion and excessive locking. Therefore, the POA Manager in TAO can be disabled.

TAO's footprint optimizations: In addition to increasing the predictability of POA request processing, omitting these features also decreases TAO's memory footprint. These omissions were done in accordance with the Minimum CORBA specification [29], which removes the following features from the CORBA 2.2 specification [1]:

- Dynamic Skeleton Interface
- Dynamic Invocation Interface
- Dynamic Any
- Interceptors
- Interface Repository
- Advanced POA features
- CORBA/COM interworking

Component	CORBA	Minimum CORBA	Percentage Reduction
POA	281,896	207216	26.5
ORB Core	347,080	330,304	4.8
Dynamic Any	131,305	0	100
CDR Interpreter	68,687	68,775	0
IDL Compiler	10,488	10,512	0
Pluggable Protocols	14,610	14,674	0
Default Resources	7,919	7,975	0
Total	861,985	639,456	25.8

Table 2: Comparison of CORBA with Minimum CORBA Memory Footprint

Table 2 shows the footprint reduction achieved when the features listed above are excluded from TAO. The 25.8% reduction in memory footprint for Minimum CORBA is fairly significant. However, we plan to reduce the footprint of TAO even further by streamlining its CDR Interpreter [15]. In Minimum CORBA, TAO's CDR Interpreter only needs to support the static skeleton interface (SSI) and static invocation interface (SII). Thus, support for the dynamic skeleton interface (DSI) and dynamic invocation interface (DII) can be omitted.

3 Optimizing the ORB Core for Real-time Applications

The ORB Core is a standard component in CORBA that is responsible for connection and memory management, data transfer, endpoint demultiplexing, and concurrency control [1]. An ORB Core is typically implemented as a run-time library linked into both client and server applications. When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. For objects executing remotely, a CORBA-compliant ORB Core transfers requests via the General Inter-ORB Protocol (GIOP), which is commonly implemented with the Internet Inter-ORB Protocol (IIOP) that runs atop TCP.

Optimizing a CORBA ORB Core to support real-time applications requires the resolution of many design challenges. This section outlines several of these challenges and describes the optimization principle patterns we applied to maximize the predictability, performance, and scalability of TAO's ORB Core. These optimizations include transparently collocating clients and servants that are in the same address space, minimizing dynamic memory allocations and data copies, and minimizing GIOP/IIOP protocol overhead. Additional optimizations for real-time ORB Core connection management and concurrency strategies are described in [30].

3.1 Collocation Optimizations

Motivation: In addition to separating interface and implementation, a key strength of CORBA is its decoupling of (1) servant implementations from (2) how servants are configured into server processes throughout a distributed system. In practice, CORBA is used primarily to communicate between remote objects. However, there are configurations where a client and servant must be collocated in the same address space [31]. In this case, there is no need to incur the overhead of data marshaling or transmitting requests and replies through a "loopback" transport device, which is an application of the principle pattern of *avoiding obvious waste*.

TAO's collocation optimization technique: TAO's POA optimizes for collocated client/servant configurations by generating a special stub for the client, which is an application of the principle pattern of *relaxing system requirements*. This stub forwards all requests to the servant and eliminates data marshaling, which is an application of the principle pattern of *avoiding waste*. Figure 12 shows the classes produced by TAO's IDL compiler.

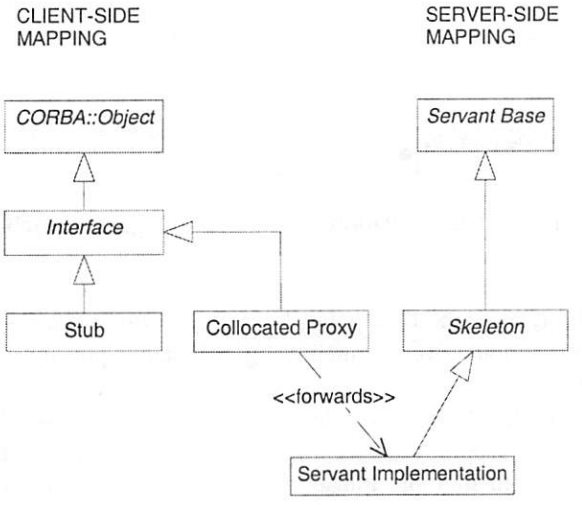


Figure 12: TAO's POA Mapping and Collocation Class

The stub and skeleton classes shown in Figure 12 are required by the POA specification; the collocation class is specific to TAO. Collocation is transparent to the client since it only accesses the abstract interface and never uses the collocation class directly. Therefore, the POA provides the collocation class, rather than the regular stub class, when the servant resides in the same address space as the client.

Supporting transparent collocation in TAO: Clients can obtain an object reference in several ways, *e.g.*, from a CORBA Naming Service or from a Lifecycle Service generic factory operation. Likewise, clients can use

`string_to_object` to convert a stringified interoperable object reference (IOR) into an object reference. To ensure locality transparency, an ORB's collocation optimization must determine if an object is collocated. If it is, the ORB returns a collocated stub – if it is not, the ORB returns a regular stub to a distributed object.

The specific steps used by TAO's collocation optimizations are described below:

Step 1 – Determining collocation: To determine if an object reference is collocated, TAO's ORB Core maintains a *collocation table*, which applies the principle of *maintaining extra state*. Figure 13 shows the internal structure for collocation table management in TAO. Each collocation table maps

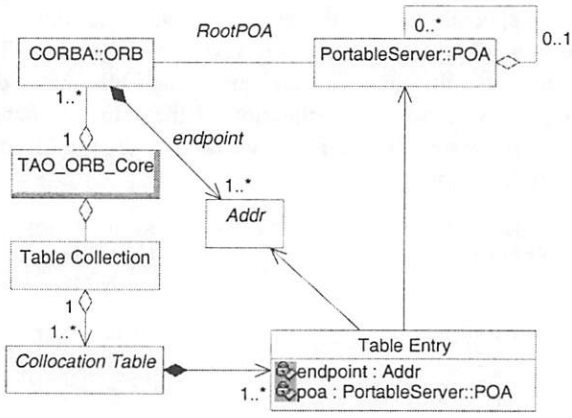


Figure 13: Class Relationship of TAO's Collocation Tables

an ORB's transport endpoints to its RootPOA. In the case of IIOP, endpoints are specified using {hostname, port number} tuples.

Multiple ORBs can reside in a single server process. Each ORB can support multiple transport protocols and accept requests from multiple transport endpoints. Therefore, TAO maintains multiple collocation tables for all transport protocols used by ORBs within a single process. Since different protocols have different addressing methods, maintaining protocol specific collocation tables allows us to strategize and optimize the lookup mechanism for each protocol.

Step 2 – Obtaining a reference to a collocated object: A client acquires an object reference either by resolving an imported IOR using `string_to_object` or by demarshaling an incoming object reference. In either case, TAO examines the corresponding collocation tables according to the profiles carried by the object to determine if the object is collocated or not. If the object is collocated, TAO performs the series of steps shown in Figure 14 to obtain a reference to the collocated object.

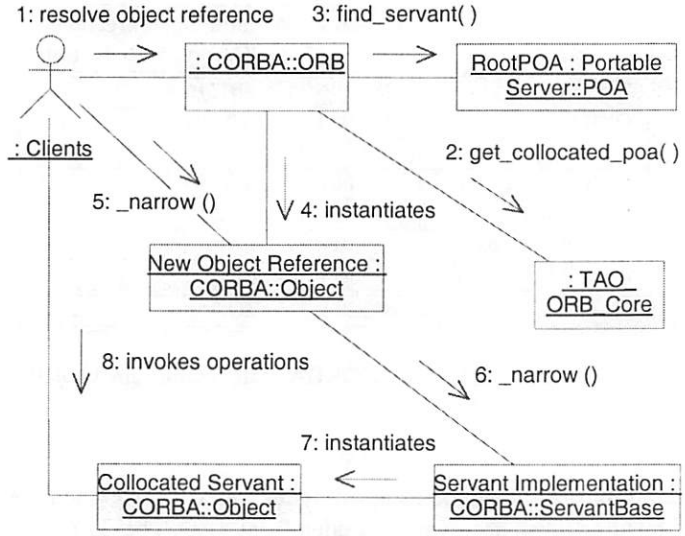


Figure 14: Finding a Collocated Object in TAO

As shown in Figure 14, when a client process tries to resolve an imported object reference **(1)**, the ORB checks **(2)** the collocation table maintained by TAO's ORB Core to determine if any object endpoints are collocated. If a collocated endpoint is found this check succeeds and the RootPOA corresponding to the endpoint is returned. Next, the matching Object Adapter is queried for the servant, starting at its RootPOA **(3)**. The ORB then instantiates a generic `CORBA::Object` **(4)** and invokes the `_narrow` operation on it. If a servant is found, the ORB's `_narrow` operation **(5)** invokes the servant's `_narrow` method **(6)** and a collocated stub is instantiated and returned to the client **(7)**. Finally, clients invoke operations **(8)** on the collocated stub, which forwards the operation to the local servant via a virtual method call.

If the imported object reference is not collocated, then either operation **(2)** or **(3)** will fail. In this case, the ORB invokes the `_is_a` method to verify that the remote object matches the target type. If the test succeeds, a distributed stub is created and returned to the client. All subsequent operations are invoked remotely. Thus, the process of selecting collocated stubs or non-collocated stubs is completely transparent to clients and it's only performed at the time of object reference creation.

Step 3 – Performing collocated object invocations: Collocated operation invocations in TAO borrow the client's thread-of-control to execute the servant's operation. Therefore, they are executed within the client thread at its thread priority.

Although executing an operation in the client's thread is very efficient, it is undesirable for certain types of real-time applications [32]. For instance, priority inversion can occur

when a client in a lower priority thread invokes operations on a collocated object in a higher priority thread. To provide greater access control over the scope of TAO's collocation optimizations, applications can associate different access policies to endpoints so they only appear collocated to certain priority groups. Since endpoints and priority groups in many real-time applications are statically configured, this access control lookup does not impose additional overhead.

Empirical results: To measure the performance gain from TAO's collocation optimizations, we ran server and client threads in the same process. Two platforms were used to benchmark the test program: a dual 300 Mhz UltraSparc-II running SunOS 5.5.1 and a dual 400 Mhz Pentium-II running Microsoft Windows NT 4.0 (SP3.) The test program was run both with TAO's collocation optimizations enabled and disabled to compare the performance systematically.

Figure 15 shows the performance improvement, measured in calls-per-second, using TAO's collocation optimizations. Each operation cubed a variable-length sequence of `longs` that contained 4 and 1,024 elements, respectively. As expected, collocation greatly improves the performance of operation invocations when servants are collocated with clients. Our results show, depending on the size of arguments passed to the operations, performance improves from 2,000% to 200,000%. Although the test results are foreseeable, they show that by using TAO's collocation optimization, invocations on collocated CORBA objects can be as fast as calling functions on local C++ objects.

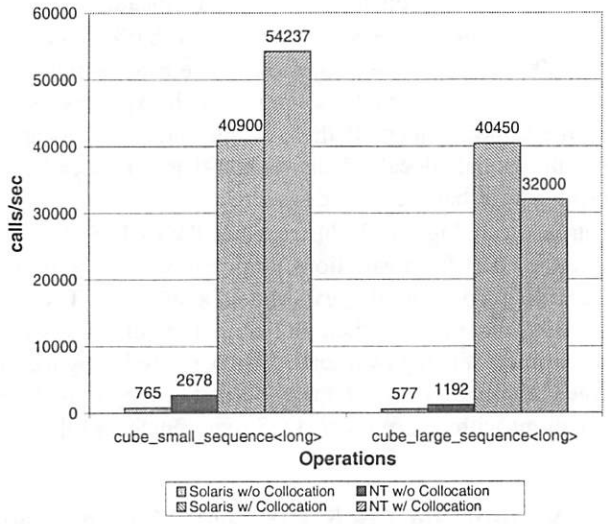


Figure 15: Results of TAO's Collocation Optimizations

TAO's collocation optimizations are not totally compliant with the CORBA standard since its collocation class forwards all requests directly to the servant class. Although this makes the common case very efficient, this implementation does not support the following advanced POA features:

- `POA::Current` is not setup
- Interceptors are bypassed
- POA Manager state is ignored
- Servant Managers are not consulted
- Etherealized servants can cause problems
- Location forwarding is not supported
- The POA's `Thread_Policy` is circumvented

Adding support for these features to TAO's collocation class slow downs the collocation optimization, which is why TAO currently omits these features. We plan to support these advanced features in future releases of TAO so that if applications know these advanced features are not required they can be ignored selectively.

3.2 Memory Management Optimizations

Motivation: A key source of overhead and non-determinism in conventional ORB Core implementations is improper management of memory buffers. Memory buffers are used by CORBA clients to send requests containing marshaled parameters. Likewise, CORBA servers use memory buffers to receive requests containing marshaled parameters.

One source of memory management overhead stems from the use of dynamic memory allocation, which is problematic for real-time ORBs. For instance, dynamic memory can fragment the global process heap, which decreases ORB predictability. Likewise, locks used to access a global heap from multiple threads can increase synchronization overhead and incur priority inversion [30].

Another significant source of memory management overhead involves excessive data copying. For instance, conventional ORB's often resize their internal marshaling buffers multiple times when encoding large operation parameters. Naive memory management implementations use a single buffer that is resized automatically as necessary, which can cause excessive data copying.

TAO's memory management optimization techniques: TAO's memory management optimizations leverage off the design of its concurrency strategies, which minimize thread context switching overhead and priority inversions by eliminating queueing within the ORB's critical path. For example, on the client-side, the thread that invokes a remote operation is the same thread that completes the I/O required to send the request, *i.e.*, no queueing exists within the ORB. Likewise, on the server-side, the thread that reads a request completes

the upcall to user code, also eliminating queueing within the ORB. These optimizations are based on the principle pattern of *exploiting locality* and *optimizing for the common case*.

By avoiding thread context switches and queueing, TAO can benefit from memory management optimizations based on *thread-specific storage*. Thread-specific storage is a common design pattern [13] for optimizing buffer management in multi-threaded middleware. This pattern allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access, which is an application of the pattern of *avoiding waste*. TAO uses this pattern to place its memory allocators into thread-specific storage. Using a thread-specific memory pool eliminates the need for intra-thread allocator locks, reduces fragmentation in the allocator, and helps to minimize priority inversion in real-time applications.

In addition, TAO minimizes unnecessary data copying by keeping a linked list of CDR buffers. As shown in Figure 16, operation arguments are marshaled into TSS allocated buffers. The buffers are linked together to minimize data copying. Gather-write I/O system calls, such as `writev`, can then write these buffers atomically without requiring multiple OS calls, unnecessary data allocation, or copying. TAO's memory management design also supports special allocators, such as zero-copy schemes [33] that share memory pools between user processes, the OS kernel, and network interfaces.

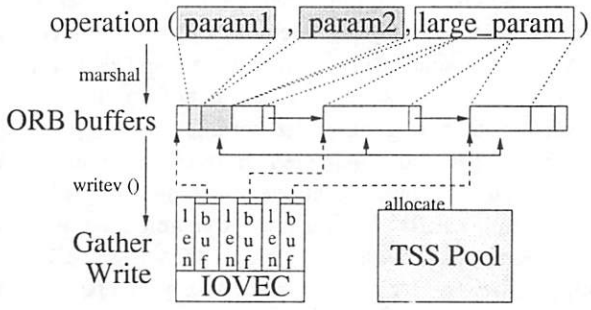


Figure 16: TAO's Internal Memory Managment

Empirical results: Figure 17 compares buffer allocation time for a CORBA request using thread-specific storage (TSS) allocators with that of using a global allocator. These experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0. The test program contained a group of ORB buffer (de)allocations intermingled with a pseudo-random sequence of regular (de)allocations. This is typical of middleware frameworks like CORBA, where application code is called from the framework and vice-versa. Both experiments perform the same sequence of memory allocation requests, with one experiment using a TSS allocator for the ORB buffers and the other using a global allocator.

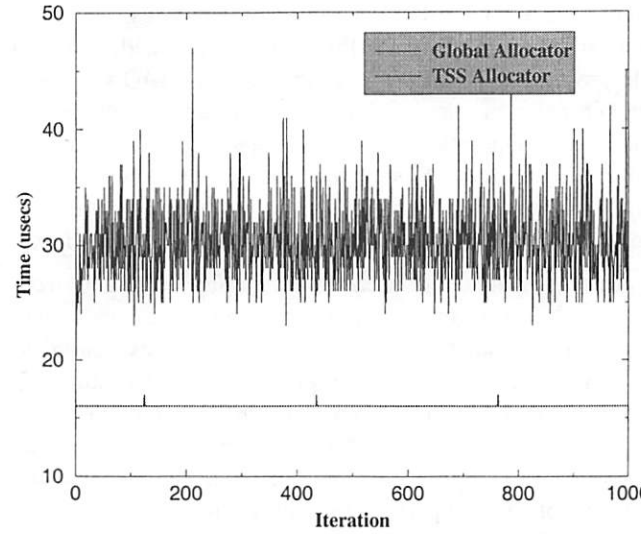


Figure 17: Buffer Allocation Time using TSS and Global Allocators

In this experiment, we perform ~16 ORB buffer allocations and ~1,000 regular data allocations. The exact series of allocations is not important, as long as both experiments perform the same number. If there is one series of allocations where the global allocator behaves non-deterministically, it is not suitable for hard real-time systems.

Our results in Figure 17 illustrate that TAO's TSS allocators isolate the ORB from variations in global memory allocation strategies. In addition, this experiment shows how TSS allocators are more efficient than global memory allocators since they eliminate locking overhead. In general, reducing locking overhead throughout an ORB is important to support real-time applications with deterministic QoS requirements [30].

3.3 Minimizing ORB Protocol Message Footprint

Motivation: Real-time systems have traditionally been developed using proprietary protocols that are hard-coded for each application. In theory, CORBA's GIOP/IIOP protocols obviate the need for proprietary protocols. In practice, however, many developers of real-time applications are justifiably concerned that standard CORBA protocols will cause excessive overhead. For example, some applications have very strict constraints on latency, which is affected by the total time required to transmit the message. Other applications, such as mobile PDAs running over wireless access networks, have limited bandwidth, which makes them more sensitive to protocol message footprint overhead.

TAO's ORB protocol optimization techniques: A GIOP request includes a number of fields, such as the version number, that are required for interoperability among ORBs. However, certain fields are not required in all application domains. For instance, the magic number and version fields can be omitted if a single supplier and single version is used for ORBs in a real-time embedded system. Likewise, if the communicating ORBs are running on systems with the same endianess, *i.e.*, big-endian or little-endian, the byte order flag can be omitted from the request.

Since embedded and real-time systems typically run the same ORB implementation on similar hardware, we have modified TAO to optionally remove some fields from the GIOP header and the GIOP Request header when the `-ORBgioplite` option is given to the client and server `CORBA::ORB_init` method. The fields removed by this optimization are shown in Table 3. These optimizations are guided by the principle patterns of *relaxing system requirements* and *avoiding unnecessary generality*.

Header Field	Size
GIOP magic number	4 bytes
GIOP version	2 bytes
GIOP flags (byte order)	1 byte
Request Service Context	$\geq$ 4 bytes
Request Principal	$\geq$ 4 bytes
Total	$\geq$ 15 bytes

Table 3: Messaging Footprint Savings for TAO's GIOPlite Optimization

Empirical results: We conducted an experiment to measure the performance impact of omitting the GIOP fields in Table 3. These experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0 in loopback mode. Table 4 summarizes the results, expressed in calls-per-second:

	Marshaling Enabled			Marshaling Disabled		
	min	max	avg	min	max	avg
GIOP	2,878	2,937	2,906	2,912	2,976	2,949
GIOPlite	2,883	2,978	2,943	2,911	3,003	2,967

Table 4: Performance of TAO's GIOP and GIOPlite Protocol Implementations

Our empirical results reveal a slight, but measurable, 2% improvement when removing the GIOP message footprint "overhead." More importantly though, these changes do not affect the standard CORBA APIs used to develop applications. Therefore, programmers can focus on the development of applications, and if necessary, TAO can be optimized to use this lightweight version of GIOP.

To obtain more significant protocol optimizations, we are adding a *pluggable protocols* framework to TAO [34]. This framework generalizes TAO's current `-ORBgioplite` option to support both pluggable ORB protocols (ESIOPs) *and* pluggable transport protocols.

4 Related Work

Demultiplexing is an operation that routes messages through the layers of an ORB endsystem. Most protocol stacks models, such as the Internet model or the ISO/OSI reference model, require some form of multiplexing to support interoperability with existing operating systems and peer protocol stacks. Likewise, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation (as shown in Figure 2).

Related work on demultiplexing focuses largely on the lower layers of the protocol stack, *i.e.*, the transport layer and below, as opposed to the CORBA middleware. For instance, [21, 35, 22, 36] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications that require real-time quality of service guarantees.

Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [37]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [38], the Mach Packet Filter (MPF) [39], PathFinder [40], demultiplexing based on automatic parsing [41], and the Dynamic Packet Filter (DPF) [36].

As mentioned before, most existing demultiplexing strategies are implemented within the OS kernel. However, to optimally reduce ORB endsystem demultiplexing overhead requires a vertically integrated architecture that extends from the OS kernel to the application servants. Since our ORB is currently implemented in user-space, however, our work focuses on minimizing the demultiplexing overhead in steps 3, 4, 5, and 6 (which are shaded in Figure 2).

5 Concluding Remarks

Developers of real-time systems are increasingly using off-the-shelf middleware components to lower software lifecycle costs and decrease time-to-market. In this economic climate, the flexibility offered by CORBA makes it an attractive middleware architecture. Since CORBA is not tightly coupled to a particular OS or programming language, it can be adapted readily to "niche" markets, such as real-time embedded systems, which are not well covered by other middleware. In this

sense, CORBA has an advantage over other middleware, such as DCOM [42] or Java RMI [43], since it can be integrated into a wider range of platforms and languages.

The POA and ORB Core optimizations and performance results presented in this paper support our contention that the next-generation of standard CORBA ORBs will be well-suited for distributed real-time systems that require efficient, scalable, and predictable performance. Table 5 summarizes which TAO optimizations are associated with which principle patterns, as well as which optimizations conform to the CORBA standard and which are non-standard.

Optimization	Principle Patterns	Compliant
Request demuxing	Precompute, Avoid waste Passing hints in header Relaxing system requirements Using specialized routines Not tied to reference models Adding extra state	yes
Object keys in upcalls	Avoid waste Exploit locality	yes
Predictability and footprint	Relaxing system requirements	yes
Collocation	Relax system requirements Avoid waste Add extra state	no
Memory management	Exploit Locality Avoid waste Optimize for common case	yes
Protocol msg footprint	Avoid generality Relax system requirements	no

Table 5: Degree of CORBA-compliance for Real-time Optimization Principle Patterns

Our primary focus on the TAO project has been to research, develop, and optimize policies and mechanisms that allow CORBA to support hard real-time systems, such as avionics mission computing [11]. In hard real-time systems, the ORB must meet deterministic QoS requirements to ensure proper overall system functioning. These requirements motivate many of the optimizations and design strategies presented in this paper. However, the architectural design and performance optimizations in TAO's ORB endsystem are equally applicable to many other types of real-time applications, such as telecommunications, network management, and distributed multimedia systems, which have statistical QoS requirements.

The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. This release also contains the ORB benchmarking test suites described in this paper.

Acknowledgements

We would like to thanks our COOTS shepherd, Steve Vinoski, whose comments helped improve this paper. In addition, we would like to thank the COOTS Program Committee and anonymous reviewers their constructive suggestions for improving the paper.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[2] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[3] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-12-05 ed., December 1998.

[4] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[6] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[7] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[8] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[9] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[10] C. D. Gill, D. L. Levine, and D. C. Schmidt, "Evaluating Strategies for Real-Time CORBA Dynamic Scheduling," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[12] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), IEEE, June 1999.

[13] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, April 1999.

[14] Alistair Cockburn, "Prioritizing Forces in Software Design," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), pp. 319–333, Reading, MA: Addison-Wesley, 1996.

[15] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1999.

[16] G. Varghese, "Algorithmic Techniques for Efficient Protocol Implementations ," in *SIGCOMM '96 Tutorial*, (Stanford, CA), ACM, August 1996.

[17] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[18] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 10, Sept. 1998.

[19] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 10, April 1998.

[20] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.

[21] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[22] Z. D. Dittia, J. Jerome R. Cox, and G. M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, (Boston, USA), pp. 179–187, IEEE Computer Society Press, April 1995.

[23] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[24] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[26] A. Gokhale, D. C. Schmidt, C. O'Ryan, and A. Arulanthu, "The Design and Performance of a CORBA IDL Compiler Optimized for Embedded Systems," in *Submitted to the LCTES workshop at PLDI '99*, (Atlanta, GA), IEEE, May 1999.

[27] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[28] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[29] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[30] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

[31] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, vol. 10, June 1998.

[32] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[33] Z. D. Dittia, G. M. Parulkar, and J. Jerome R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), IEEE, April 1997.

[34] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Submitted to the IFIP* 6^{th} *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

[35] D. C. Feldmeier, "Multiplexing Issues in Communications System Design," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 209–219, ACM, Sept. 1990.

[36] D. R. Engler and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, (Stanford University, California, USA), pp. 53–59, ACM Press, August 1996.

[37] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: an Efficient Mechanism for User-level Network Code," in *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[38] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[39] M. Yuhara, B. Bershad, C. Maeda, and E. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," in *Proceedings of the Winter Usenix Conference*, January 1994.

[40] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson, "Pathfinder: A pattern-based packet classifier," in *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*, USENIX Association, November 1994.

[41] M. Jayaram and R. Cytron, "Efficient Demultiplexing of Network Packets by Automatic Parsing," in *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, (University of Arizona, Tucson, AZ), February 1996.

[42] Microsoft Corporation, *Distributed Component Object Model Protocol (DCOM)*, 1.0 ed., Jan. 1998.

[43] Sun Microsystems, Inc, *Java Remote Method Invocation Specification (RMI)*, Oct. 1998.

The Application of Object-Oriented Design Techniques to the Evolution of the Architecture of a Large Legacy Software System

Jeff Mason (jeff.mason@xilinx.com) and Emil S. Ochotta (emil.ochotta@xilinx.com)
Xilinx Inc.
2100 Logic Drive
San Jose, CA 95124

ABSTRACT

Object Oriented Analysis and Design (OOAD) is increasingly popular as a set of techniques that can be used to initially analyze and design software. Unfortunately, OOAD is a relatively new concept and many large legacy systems predate it. This paper presents the approach one company followed in applying OOAD techniques to an existing 2.5 million line code base. We present an iterative process that provides an avenue for the software to evolve while balancing the needs of business and software engineering. Our case study reveals the many pitfalls that can derail a software re-engineering effort, but also shows promising initial results from continued perseverance in this effort.

1. Introduction

Object Oriented Analysis and Design (OOAD) techniques promise many benefits to software developers and software companies - software reuse and resilience to change through component libraries and patterns[1][2], lucid code structure that more clearly reflects the problem domain[3], and reduced risk by introducing a formal design process where often none existed previously[4] - to name only a few. To reap these rewards, most OOAD techniques assume software developers apply the techniques at the beginning of the software lifecycle, i.e., the beginning of the design process, and continue to use them as the software matures. Unfortunately, OOAD is a relatively new concept and many large legacy systems predate it. Moreover, because the pressures of commercial competition focus directly on adding features, fixing bugs, and releasing the product on-time, software developers often (misguidedly) skimp on the things that should be done for long term benefit in favor of the things that absolutely must be done to complete the product. Since not all developers are educated as to the benefits of OOAD it is often one of the things overlooked in the headlong rush to a software release. The long-term price of this behavior is a large body of difficult to maintain software, proving the well-known adage that the overriding cost of software is not its initial development but rather its maintenance. In this environment of legacy software and corporate pressure, reaping the benefits of OOAD seems a very elusive goal.

This paper describes the process one company undertook to re-architect their large legacy software system and begin reaping the benefits of OOAD techniques despite the constraints of continuing feature improvements and a strict release schedule. This six-step process is as follows:

1. *Analysis*: evaluating the current state of the legacy software;
2. *Goal Selection*: determining a set of goals to guide changes to the software and allow evaluation of the results;
3. *Key Concept Selection*: refining the goals into a set of key concepts based on business requirements, software engineering principles and object oriented analysis and design principles;
4. *Planning*: determining how best to apply the key concepts to the legacy software to allow it to evolve towards a system that satisfies those concepts;
5. *Implementation*: making it happen; and
6. *Measurement*: evaluating the effectiveness of the changes against the original goals.

There is a substantial body of research that focuses on the technical aspects of software evolution[5][6] and reengineering[7][8], and many of the technical ideas discussed in this paper have been described elsewhere in one form or another. The contribution of this paper is the description of the process we undertook and how we selected and satisfied key concepts that balanced the demands of business, the requirements of software engineering, and the OOAD principles we wanted to pursue. In the end, these key concepts included:

- ***Autonomy***: encapsulating and insulating functionally related software into subsystems to minimize interactions, to reduce compile times, and to support testing, allowing these subsystems to evolve independently and asynchronously;
- ***Sharing:*** solving problems in as few places and as few times as possible to maximize code reuse, minimize code size, and promote standardization;
- ***Comprehensibility***: promoting design, documentation and coding standards that - for the general client - make shared code and interfaces easier to understand, more convenient to use, and easier to maintain;

- ***Modularity***: allowing functional product components to be released to end users independently and asynchronously;
- ***Co-development:*** promoting the ability to explore, evaluate, and develop new features without affecting other on-going development;
- ***Innovation***: promoting runtime, memory, and quality of results performance through optimization and innovation;
- ***Testing:*** enabling efficient automated testing by creating a levelizable system[9] (i.e., a system where the testing and compile-time dependencies between software modules form a directed acyclic graph);
- ***Release***: supporting a release model with fixed release dates planned long in advance.

To implement these concepts, we developed a *system architecture vision* that outlined the changes to the software architecture that were designed to put these key concepts into practice. We then put forward an evolutionary plan to implement the vision. What quickly became apparent is that the inertia of the software was too large to allow all our changes to be implemented at once, while still releasing working software on an aggressive fixed schedule. Consequently, we realized that the six-step process described above must be applied iteratively, over an extended period of years.

Since the full implementation of this vision is an ongoing task whose costs and benefits may not be fully evaluated for many years, this paper describes the initial iteration through that six-step process. In this first iteration the implementation had to be scaled back to fit within a single release cycle of less than a year and focussed primarily on the key concepts of autonomy, sharing, testing, and comprehensibility. In these areas, we have seen some dramatic improvements, particularly where quantitative measurement is straightforward, such as compile-time coupling.

The remainder of this paper is organized as follows. In the next section we detail the first step in the six-step process we followed, outlining the state of the software system and the corporate situation that forms the backdrop for our work. In Section 3 and Section 4, we describe the next two steps in the process, the conceptual steps of setting the correct goals we are working toward and selecting key concepts that reflect those goals. In Section 5, we present the evolutionary plan we created to work towards realizing those key concepts in our software. In Section 6, we discuss the implementation of this evolutionary plan, and in Section 7, we evaluate this implementation against the key concepts and our initial goals. Finally, in Section 8, we present our conclusions.

2. Background and Analysis

In this section, we describe the first of the six steps in the process we followed to re-architect our legacy software system. We first present the environment in which our work was performed, including a brief description of Xilinx, the company where the work was performed, and the purpose of the software. We then discuss the state of disrepair we found when we first began to look at the software system itself and the costs associated with that disrepair. These costs were the initial motivation that drove our re-architecture.

2.1 Xilinx Inc.

This work was performed at Xilinx[10]. Xilinx was created as a hardware company, producing FPGAs, which are members of the family of integrated circuits (ICs) called programmable logic.

Understanding how an FPGA is used provides some useful insight into the complexity of the FPGA design software we discuss in this paper. An example FPGA application is emulating another IC or computer chip. In this application, the design to be emulated is loaded into the FPGA and the FPGA inserted into the system of which the chip being emulated is a part. This technique allows the design of the new chip and the system of which it is part to be tested and debugged before the new chip is actually built. Similar to a compiler, FPGA design software automatically translates the high-level description of the chip to be emulated into millions of programming bits that configure the FPGA to perform the emulation. Part of this translation task involves selecting a location from among the thousands available on the FPGA for each logical element. These locations must be selected to optimize chip performance or other user-specified constraints, creating an NP-complete[11] combinatorial optimization problem[12]. Moreover, in response to competition and customer demand, FPGAs are continually increasing in size and new hardware features are added to each new FPGA[13]. To keep pace with these newer, bigger FPGAs and still provide new software features, the FPGA design software is increasing in size and complexity at an even faster rate. Finally, because software provides the abstract model with which most FPGA customers interact, Xilinx has put increasing emphasis on software development in order to turn our software into a competitive advantage. The difficulty of the FPGA optimization problem, continually evolving FPGA hardware, and increasing customer reliance on fast reliable software conspire to make writing FPGA design software a challenging proposition.

2.2 The State of Xilinx Software

As one step toward improving its software, in early 1995 Xilinx acquired a small software start-up company based in Colorado. At that time, Xilinx' FPGA design software consisted of nearly 1.5 million lines of C code developed and maintained by approximately 70 staff members. Xilinx had released 30 software revisions to over 10,000 software customers. In contrast, the 30 engineers of the close-knit start-up had written just over 700,000 lines of highly interconnected C++ and had

released 6 software revisions to about 200 customers. The start-up's code was poorly documented, but a knowledgeable person was always at hand to deal with any issue or problem. Thus, change requests were informal conversations and system-wide changes could be implemented and compiled within a few hours.

After the purchase, the corporate goal was to merge the two software systems, keeping the strengths of both. This was easier said than done. The C++ code from the start-up was selected as the software base for the future merged product, and features that had been added to the original Xilinx product based on customer requests were to be added as needed. Software developers in Colorado were now faced with a much larger development environment and had to work with developers in California who understood the features to be added but did not understand the software base. Software developers in California were now faced with giving up their old software, learning a new and undocumented software base and working with developers in Colorado who did not understand the new features to be added. Neither group was used to working across multiple development sites, so "lack of communication" was one of the most common complaints by both groups about their peers on the other side of the mountains. Software was not getting built on time and fingers were being pointed in all directions. It was a difficult time for all involved.

Work toward the first merged release took substantially longer than anyone had dared to predict, and we missed several target release dates. Upper management began to apply greater pressure to the software team, justifying decisions to take "short-cuts" on the basis of short-term necessity. As is frequently the case, it is arguable whether these "short-cuts" reduced the time to first customer shipment, but they unquestionably came back to haunt us by adding to our maintenance burden over the next few releases.

After the frenzied days and nights of making our first few merged releases a reality, we took stock of our new software. The start-up's 700,000 lines of C++ had ballooned to approximately 2.5 million lines of C++ code in roughly 2200 source and header files. Our software shipped as 45 executables, 130 shared libraries (loaded at program start up), and 110 dynamically loaded libraries that customized the software for the different FPGAs in the Xilinx product line. Our single source software supported the Solaris, Windows, HP and RS6000 platforms. The source code was organized into approximately 400 subdirectories called packages, where each package produced either a library or an executable. After a brief inspection, we identified several major problem areas that we later categorized according to the key concepts to which they relate:

- **Comprehensibility**. Just as it was in the start-up, the code was mostly undocumented, but now it was much more complex and growing so rapidly that it was no longer possible to find any one person who understood most of it.
- **Autonomy (Encapsulation).** The interfaces between packages had evolved as necessary to meet tactical, local needs, without regard for strategic, system-level concerns. Consequently interfaces were extremely broad and ill defined. There was no clear division between the interface and the implementation of most classes. Much of the code was really just old C code transformed into C++ objects. One of the major indicators of a lack of encapsulation was direct access of class data by another class. Many of our classes had been designed with public get/set functions for each of the class data members. Consequently, changes that should have been internal to a package had repercussions throughout the system.
- **Autonomy (Insulation).** The compile-time dependencies (due to included files) had never been designed or analyzed. Often the vast majority of the compilation time for a module consisted of reading and processing included files. When first designing C++ classes, the tendency is to make the header file as convenient as possible for the implementation of that header. For example, the lowest level header file in the Xilinx software system directly or indirectly included almost 60 system header files, establishing a platform independent interface to the operating system. However, in such a large system, most of this functionality was not used by most of the clients that included it. This overhead is an unneeded burden to clients, who often compile complete definitions of many unused classes or classes that require only a forward reference. Engineers, recognizing this system-wide problem but unable to change it, were starting to make extremely large source files because the compilation times were faster than the aggregate compilation time of many smaller files.
- **Autonomy (Insulation).** Another problem was the rampant use of 'inline' functions. Inline functions are expanded at compile time rather than run time. This implies that a class that defines an inline function can not change the implementation of that inline function without forcing all of its clients to recompile.
- **Autonomy.** The turn around time to build and verify our software had become one to two weeks. Most of that time was spent in tracking down integration problems and then rebuilding everything. Because of the interdependence of the software, a compilation problem in one package may actually be caused by interface problems in any one of a large number of packages. Tracking down and solving these integration problems was made even more difficult because finding someone who understood the disparate parts of the system was no longer possible. Because of the difficulty of compiling several million lines of code on a single workstation, developers typically developed and tested against builds that were several

weeks out of date, exacerbating the integration problems for the next build.

- **Sharing and Autonomy.** There was no person or group whose responsibility it was to review or coordinate code changes. Each engineer or group was free to implement or use what they needed to get their specific job completed. Sometimes system-wide integration builds failed because large changes were made to shared code to support a new feature, but the changes were not tested for all clients. Other times, when small changes to a large package were required, engineers would copy the entire package into their package to avoid having to work with the other package's owner.

Problems like these were creating a software and corporate environment where developers no longer had the freedom or time to innovate. They had no freedom because every non-critical project was deemed high risk, since the complex package interdependencies could cause minor errors to have major repercussions throughout the system. They had no time because fixing each small problem required an inordinate amount of time to implement and verify.

After this analysis, it was clear that something needed to change. Fortunately for Xilinx, senior management understood the issues and that the long-term viability of the software product was at stake. With their support, several members of the company were chartered with re-architecting the software to fix these problems.

3. Goal Selection and The System Architecture Committee

The software management team recognized that Xilinx' software needed significant re-design at the architectural level, requiring co-operation from the entire software organization. They created the System Architecture Committee, a seven member team of engineers and managers that represented both development sites. The VP of software was a member of the committee, giving it the needed management clout. The authors were selected as members of this committee.

Initially, it was thought that members of the committee would spend roughly 10% of their time looking at system architecture issues, but as the weeks passed and the extent of the problem became more clear, the workload quickly grew beyond 10% of each member's time. To give the architectural work the attention it required, the authors became full time architects, and most members were required to put aside their other duties for short periods of time to complete work for the committee.

In the first few weeks of meetings, little was accomplished and frustrations grew. Several members proposed changes that they felt would improve the existing software architecture, but the group could not reach consensus. Eventually, it became clear that the goals of the various members of the committee were inconsistent, which led to disagreement over the changes that were required, which in turn led to stalemate and inaction. Consequently, the committee had to agree on its goals before it could take any steps to improve the software architecture. Choosing the goals was the seed for the six-step process that we eventually followed to bring our architectural changes to fruition.

The committee agreed upon six goals, several of which conflicted, making them impossible to satisfy all the goals simultaneously. Initially the group was disheartened that we could not select a set of goals that could be satisfied completely, but over time it became clear that this tension between the goals reflected the reality of business and of the software design process. In both environments, there are no right answers and compromise is essential to success. Moreover, the ability to strike the correct balance between competing goals is what distinguishes successful businesses and software organizations, and this made the design process challenging and exciting.

After much discussion and negotiation, the committee agreed upon the following six goals:

- *Provide superior end user productivity.* Make internal architectural improvements that eventually result in customer visible improvements in our software. Xilinx customers are the first priority.
- *Distribute productivity effectively across development groups.* Address "geography problems," where developers in different groups do not communicate. The developers who were originally in the start-up felt they could not get their work done due to continually having to educate the other developers. The other developers felt they could not get their work done because they were not trusted to modify the existing core of the software. In practice, geography problems can happen even when the groups are physically adjacent, and the software architecture can have a significant effect on inter-group communication. These problems have a large negative impact on productivity and morale.
- *Improve the productivity of individual developers.* Create an environment where individual contributors can work more efficiently, without having to wait for other developers to complete their tasks.
- *Enable parallel development targeting multiple release dates.* Develop an environment that supports projects that require more time than a single release cycle. This goal is a direct result of the fixed release schedule required to support new FPGAs in a timely fashion.
- *Build in flexibility to handle a constantly changing market.* Anticipate the aspects of the software that will most likely change: new kinds of FPGAs, new software features, etc. Ensure that the software architecture is not brittle when these kinds of changes are required.

- *Enable accurate and efficient measurement of the quality of the system by designing for testability.* Plan from the outset to incorporate a testing infrastructure that supports measurement of software quality that is both fast and accurate.

These six goals formed the foundation for the rest of our software re-architecture work. They were driven primarily by business rather than OOAD or software engineering goals. When creating them, we also explicitly decided *not* to consider how we would accomplish these goals. They are merely what we wanted in an ideal world. Consequently, they form an ideal set of metrics with which we can evaluate the efficacy of our software architecture decisions.

4. The Key Concepts

Once the goals were in place, the next step was to determine how to achieve them. We soon realized that there was too large a semantic leap from the goals to actual architecture and code changes. What was needed was an intermediate step where we agreed on a set of principles from the worlds of OOAD and software engineering. These principles would reflect the above goals but more closely relate to the software and code architecture itself. This tighter relationship to the software would make it possible to create an implementation plan.

These principles are the eight key concepts introduced in Section 1 that tie our architecture work together. The first two (autonomy and sharing) are primarily OOAD techniques, and the last (release) is a business constraint. The others lie somewhere on the spectrum of OOAD techniques and plain old software engineering. As with the goals, these concepts are in tension: any plan will favor some concepts over the others. In this section we describe these key concepts and how they connect the six goals to changes that can be realized in a software architecture.

4.1 Autonomy

- **Autonomy**: encapsulating and insulating functionally related software into subsystems to minimize interactions, to reduce compile times, and to support testing, allowing these subsystems to evolve independently and asynchronously.

Autonomy follows directly from both of the productivity goals: *distribute productivity effectively across development groups* and *improve the productivity of individual developers*. Engineers are most efficient when they are free from dependencies and allowed to work alone or as members of a small, tightly-knit group.

Software dependencies can be exacerbated by poor software architecture and by failing to adhere to OOAD basics. For example, failing to encapsulate a data structure means that clients of a package use that data structure directly. When the data structure changes, the client must also change. This is an example of poor autonomy, since both the client and the supplier may be forced to wait for each other. The supplier may not be allowed to change the data structure until the client is ready, or the client may be unable to compile code that requires the new data structure until the supplier completes its implementation.

We recognize two facets of autonomy that are closely linked to OOAD principles: insulation and encapsulation. Insulation can be defined as the process of avoiding or removing unnecessary compile-time coupling[9]. In practice, insulation can be implemented by creating an opaque interface. For example, Lakos defines a fully insulated class as one that is not derived from another class, contains no inline functions or default arguments, and contains only a single pointer to an implementation class that is declared with a forward reference. The details of the implementation class are completely hidden from any clients that include the fully insulating class. The effect of full insulation is to create header files that are completely independent of each other, dramatically reducing the compile-time overhead of header file inclusion.

Another facet of autonomy is encapsulation, which should be familiar to practitioners of OOAD. Encapsulation can be defined as the concept of hiding implementation details behind a procedural interface[9]. Encapsulation and insulation are clearly related, but a fully insulated class need not be encapsulated. For example, a fully insulated class can still expose its implementation by providing public access functions to all its private data. However, in some respects encapsulation can be a less drastic technique than insulation because encapsulation allows the use of other features of C++, such as inheritance and inline functions. In this paper we refer to insulation when discussing the compile-time independence of modules from one another and refer to encapsulation when discussing the logical independence of a client class from the implementation decisions of its suppliers.

4.2 Sharing

- **Sharing**: solving problems in as few places and as few times as possible to maximize code reuse, minimize code size, and promote standardization.

Sharing falls into the general category of software reuse, a subject frequently discussed in the literature (see for example[15]). Reuse or sharing is also connected to the goal of developer productivity because in principle it allows a piece of code to be written once and reused in several places. In practice, sharing is difficult to achieve because the clients of the shared code must agree on what exactly the code does. If the code is too specialized, it is unlikely to be useful to more than one client. On the other hand, if the code is too general, it will be too slow or so simple that reusing it accomplishes little.

For the Xilinx software system, two kinds of sharing or reuse are of particular interest. Because Xilinx supports a number of different hardware devices that are fundamentally related, the Xilinx software is an excellent candidate for sharing via domain engineering[16]. In domain engineering, tasks that are needed throughout the domain are abstracted and written once. In this case, common tasks needed to support all devices can be abstracted and written as configurable or data driven algorithms. Fortunately, this characteristic had been recognized by the original designers of the core software created in the start-up and the software already made significant use of this kind of sharing (although the term domain engineering had yet to be coined).

The second kind of sharing was not as well supported in the Xilinx software, and that is the more conventional sharing of small generic algorithms. In sharing of this kind, tasks that are not domain specific, but may be general mathematical functions, data structures, or other algorithms are collected into a reusable library. To succeed at this sharing, this library has to be carefully designed explicitly so that it can be reused. The designers have to pay particular attention to making the functionality general, efficient, and well documented.

4.3 Comprehensibility

- **Comprehensibility**: promoting design, documentation and coding standards that - for the general client - make shared code and interfaces easier to understand, more convenient to use, and easier to maintain.

Comprehensibility as defined by this key concept is not intended to increase the amount of communication between developers, but to reduce the need for it. This key concept again relates back to the productivity goals. The idea is to create a system and an environment that inherently reduces the need for additional documents to describe the architecture of the system itself. One of the main benefits of such a system is the reduced need for maintenance that can occur when a change to an interface must be made both in code and in one or more separate documents.

4.4 Modularity

- **Modularity**: allowing functional product components to be released to end users independently and asynchronously.

Modularity is related primarily to the goal of *superior end user productivity*, but is an existing strength characteristic of the Xilinx software. As an example of this key concept, software support for a single Xilinx device could be shipped as part of the overall software system or as an individual software plug-in. This modularity made it possible to create and support new hardware products without shipping a complete new software system.

4.5 Co-development

- **Co-development**: promoting the ability to explore, evaluate, and develop new features without affecting other on-going development.

The key concept of co-development has two aspects that relate to what is being developed concurrently. Both relate to the goal of flexibility in a constantly changing market. In the case of support for new hardware devices, co-development means that new hardware can be supported with a minimal impact on software. This is essential in a competitive marketplace where the most successful company is the one that can innovate and respond to change the most quickly. Similarly, the other aspect of co-development is support for features and changes that are not driven by hardware, but must be developed somewhat independently from the main body of software because they extend beyond a single release cycle.

4.6 Innovation

- **Innovation**: promoting runtime, memory, and quality of result performance through optimization and innovation.

The innovation concept follows directly from the goal of providing superior end user productivity, which is fundamentally tied to software performance. Superior performance can be achieved using two methods that are in tension. The first method is optimization. This can be thought of as tuning existing software to improve its runtime, memory, or quality of result performance. Tuning software can sometimes compete with OOAD design principles such as encapsulation. For example, exploiting the underlying implementation of a data structure can sometimes result in significant improvements in performance, but at a clear cost in encapsulation.

The second method to improve performance is algorithmic change. For example, a developer may be able to squeeze a few percentage points of improvement out of a bubble sort algorithm by changing array operations to pointers and making function calls in-line. However, changing to a quick sort will yield significantly greater runtime improvements for large datasets because quicksort has better algorithmic complexity.

The two methods of improving performance are in tension because detailed optimizations that increase coupling of client algorithms to supplier algorithms also make it extremely difficult to innovate by changing either the client or the supplier algorithm. In most cases, algorithmic innovation yields greater improvements than optimization, so the focus of this key concept is on enabling innovation.

4.7 Testing

- **Testing**: enabling efficient automated testing by creating a levelizable system[9] (i.e., a system where

the testing and compile-time dependencies between software modules form a directed acyclic graph).

The testing concept corresponds directly to the testability goal. In this case, the concept has a technical definition that can be concretely evaluated. By building a graph from the compile-time dependency structure, the system can be evaluated to see if it is levelizable. If there are any loops in the dependency graph, the system is not levelizable and is more difficult to test. This is because all modules involved in a loop must be tested together as a single unit. In the worst case, all modules will be involved in a loop and the entire system must be treated as a monolithic black box for testing. Since the difficulty of testing a module grows exponentially with the size of the module, creating a levelizable system is a desirable property. In a large system such as the Xilinx software system, it is easy to accidentally create a dependency that creates a loop in the compile-time dependency graph. The size of the system also makes such loops especially expensive in testing time.

4.8 Release

- **Release**: supporting a release model with fixed release dates planned long in advance.

The release concept is closely tied to the goal to *enable parallel development targeting multiple release dates.* An additional aspect of the release goal is to force the development to happen gradually in an evolutionary fashion. By requiring customer releases on a fixed schedule, the development is forced into an evolutionary path, which reduces schedule risk.

In summary, the creation of these goals and key concepts was a long and arduous process. However, because the key concepts provided techniques to realize the goals, subsequent work went significantly faster. Each new idea could be readily compared with the goals and concepts we had already agreed to implement, helping to keep the re-architecture process on track.

5. Planning

Armed with the newly created sets of goals, the key concepts, and a common mindset, the system architecture committee began to look at the software and come up with concrete plans for what should be changed. Here again, we followed a process that is clear in retrospect but at the time seemed full of bumps and blind alleys. We began by evaluating the current architecture against the goals and key concepts. We then chose the key concepts to be given first priority in the redesign effort. Based on the highest priority key concepts we proposed several different architectural solutions intended to address these concepts, then collected the best features of these proposals into a coherent document called the *system architecture vision*. With the vision as an endpoint, we created a plan to evolve from the starting point of our existing software architecture. Finally, we imposed the constraints of having to support new FPGAs, add new software

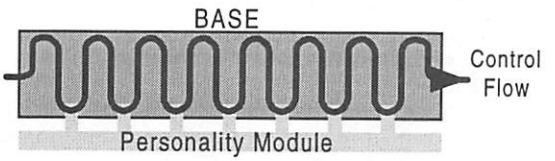


Figure 1. The Personality Module (PM) contained device-specific code that plugged in to the base code, but control flow was determined by the base.

features, fix bugs, and work with limited resources and a fixed release date. Considering these constraints, we then extracted a detailed short-term plan that would get us through the current release cycle. In this section we describe each of these planning phases in greater detail.

5.1 Prioritizing the Key Concepts

Before we could come up with an implementation plan of attack, we first needed to prioritize among the key concepts and decide which of them needed the most attention. To do this, we performed a careful evaluation of the existing software architecture against the goals and key concepts we had labored over for so long. This task allowed us to see which of the key concepts was least supported in the current software, and then to decide how we should focus our redesign efforts.

Based on this analysis, *autonomy* emerged as the most important of the key concepts to guide changes to the system. Secondary emphasis was given to *sharing*, *testing*, and *comprehensibility*. This ordering did not discount the importance of the other concepts - indeed the final solution would need to balance among all eight - but it recognized that the existing architecture already had certain strengths. The existing source code architecture was composed of two levels of hierarchy. The first level, called the Personality Module (PM) reflected the hardware device supported by that part of the software. Each PM contained packages, grouping the software within a PM by logical function. The remainder of the code, shared by all PMs, was called the "base". This organization inherently supported sharing and re-use of base code by all other PMs. Moreover, each package from a PM created a Dynamically Loaded Library (DLL) that was loaded on demand, once the base code determined the device and required functions. As shown in Fig. 1, the DLL for the PM plugged into the base software, customizing it for a given hardware device. This meant that if the base required no changes an entire PM could be developed independently of the rest of the system (the co-development concept) and shipped to customers separately from the rest of the software (the modularity concept). Finally, much of the tight coupling in the system was done in the name of performance optimization. This tight-coupling was a two edged sword however, allowing significant performance gains via detailed optimization on one hand but on the other hand stifling the creation of new

algorithms that promised leaps in performance. Here again we thought that increased autonomy was the key as it could increase encapsulation and make it easier to innovate algorithmically.

A secondary focus was the need for additional sharing. As already described in Section 4.2, the combination of base and personality module was an ideal situation for domain engineering, so there was significant sharing because the base code was reused for every device. However, there was no natural place in the system for algorithms and data structures that did not belong to any particular PM, yet did not define a new application for the base. Additional sharing of these generic algorithms was a secondary consideration for the architectural redesign.

Significant additional improvement was also desired in the area of testing. Within the existing architecture, anything not in a PM was added to the base, resulting in a very large base. Within the base, there were no rules about compile-time dependence and several packages were involved in compile-time loops. Also, we determined that significant gains in testability could be achieved by re-factoring the software according to function and designing from the outset a system that could be tested incrementally in sections.

Finally, we sought improvement in comprehensibility. Because the system had evolved into more than 400 packages, it was impossible to find a single person who had even a cursory understanding of the role of each package in the system. This made learning the system difficult for new developers, made tracking down integration problems difficult, and made it almost impossible to consider any large scale decisions about system structure.

5.2 The System Architecture Vision

Based on our priorities for the key concepts, we began to create a vision of where the software should be headed to better address autonomy, sharing, testing, and comprehensibility. This vision contained several elements and extended into the far future. Consequently, only two of these elements played a significant role in this first iteration through the six-step architecture re-design process. These elements were the re-organization of the source code into subsystems and layers and the creation of a special layer for generic algorithms. This section discusses these concepts in further detail.

Many long hours of discussion went into the creation of the vision document. After failing to write anything collectively, we delegated the task of an initial vision to one committee member. Having this draft allowed us to work through many problems and refine the concepts. After several iterations we completed an initial draft. We further refined the document based on feedback from a group of the top engineers in the software organization. Finally, the first version of the system architecture vision document was published and presented to all engineers.

However, the planning work did not stop there. The vision encompassed work that could take years to complete, but the next release was less than one year away. Moreover, with each release we had to support the latest hardware devices and offer improvements in features, runtime and software quality. After many hours of negotiation with marketing, sales, application support and senior management we reached a consensus on the resources that could be spent on re-architecting the software system. Matching available resources against the system architecture, we determined that we could make two major architectural changes: the addition of support for generic algorithms and the re-structuring of the software into layers and subsystems. Of these, the re-structuring of the software was a significantly larger investment. We describe these two changes in turn.

5.2.1 Layers and subsystems

Re-structuring the system into subsystems and layers called for a complete re-organization of the system from PMs and packages. It called for the creation of new units of functionality called subsystems, which would in turn be collected into layers.

On the surface, the software was still organized into a two-level hierarchy. However, the subsystems were envisioned as very different from the packages they replaced. These differences included the following:

- A Subsystem is typically larger than a package and can produce multiple libraries or executables. The structure within a package was flat, but a subsystem is truly hierarchical.
- Subsystems are logically related pieces of code that can have multiple people working on them. A single subsystem contains the base code and all the PM code for a single application or function.
- A subsystem provides a single directory that contains the *subsystem interface*, *i.e.*, any files exported by that subsystem, including header files, data files and libraries. Other subsystems can access only those files explicitly exported.
- Developers are encouraged to encapsulate and fully insulate their subsystem interface.
- New code in subsystems must follow a naming convention that limits pollution of the global namespace.
- Subsystems can have a compile-time dependency upon another subsystem only if that subsystem is in the same layer or a layer listed as a supplier layer. The graph of compile-time dependencies within a layer must not contain any loops.
- Part of the subsystem interface is a subsystem definition document that describes the subsystem and each exported header file. To avoid synchronization prob-

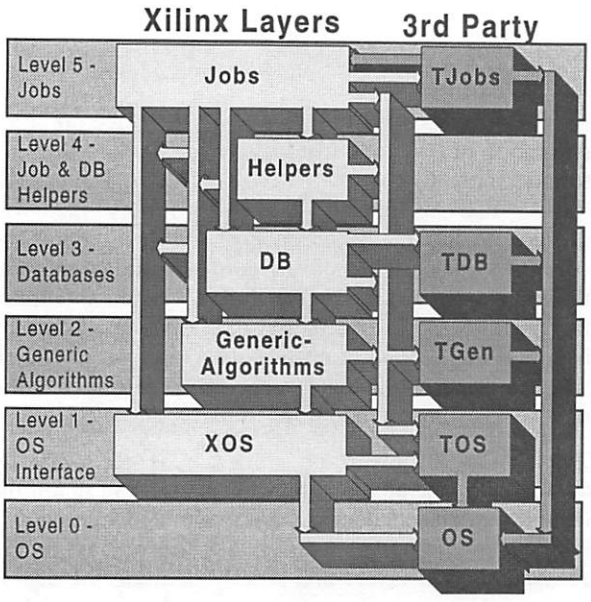


Figure 2. Include file relationships of Layers as put forward in the SAV document.

lems, the header file documentation is generated from the source code by ccdoc[14].

- The interface to a subsystem is controlled and changes require the approval of a committee. Synchronization of the changes within subsystems is handled differently than synchronization of changes in the subsystem interface. Each subsystem has a subsystem integrator, a new role that makes sure the different engineers working on the subsystem communicate. By having a single person responsible for each subsystem we expect to eliminate some, but not all, of the final build integration problems.

Of the changes proposed in the system architecture vision, these changes were perhaps the most directly connected with OOAD techniques.

Subsystems are grouped into layers. A layer is a set of subsystems with certain compile-time dependency or access rules. All subsystems within a layer have to abide by the access rules of that layer. As shown in Fig. 2, the entire Xilinx software system is composed of ten layers. Five layers contain code that originates within Xilinx, and the other five layers contain code that originates outside of Xilinx. In the figure, the arrows show access. For example, of Xilinx subsystems, only those in level 1 can directly include system header files. The principal purpose of access rules is to improve testability. However, the access rule that limits access to the system header files has the added benefit of making it easier to port to a new development platform.

5.2.2 Generic Algorithms

The second change to the software architecture is also apparent in Fig. 2. This is the addition of the generic algorithms layer. The generic algorithms layer encourages code sharing and reuse for numerical algorithms, data structures, and other generic algorithms. In addition to the code sharing between base and PM code (now contained within a subsystem) the intent was that the generic algorithms could be used for varied tasks throughout the software.

5.3 How the Architectural Vision Implements the Key Concepts

Recall that for this first iteration of improvements to our software architecture, we focussed on the key concepts of autonomy, sharing, testing, and comprehension. To work toward better support for these concepts, we chose to implement two major changes from our system architecture vision: a generic algorithms layer and the re-organization of the system into subsystems and layers. In this section we describe how the four key concepts on which our re-architecture effort focussed were realized through these two major architectural changes.

5.3.1 Autonomy

Since our engineering organization was split between two distinct sites, and a number of remote engineers were also involved, we felt that a rearrangement of our software along functional lines could help us provide better support for autonomous code development. Repackaging the software into subsystems created modules that are more self-contained and can be worked upon independently of other pieces.

Where modules are dependent on one another, greater stability is ensured by encapsulation, insulation, and committee approval for interface changes. Although some engineers felt these restrictions on the interface were overly harsh, we decided the best way to control the overall software structure was to control the subsystem interfaces. Ideally, we would also guarantee that the exported functionality behind the interface was also stable. For example, we could try to ensure that the return codes for a function do not change. However, in practice this is impractical and we rely on engineers to handle this level of detail.

Encouraging developers to use encapsulation and insulation techniques for their subsystem interface was a direct step towards improving those aspects of autonomy. As discussed in Section 4.1, complete insulation would forbid techniques such as inline functions, inheritance and default arguments[9]. We performed some simple tests and found that in certain cases changing a small set of inline functions to be called functions could cause a significant run time penalty. Since several of our applications can run for many hours, this performance penalty was not acceptable. Similarly, we have a reliance on inheritance that is used as the principal mechanism for writing code in dynamically loaded libraries. Because of these constraints, we could not require full insulation for all

subsystem interfaces, but left it initially to engineer's discretion, subject to external review.

We further limited compile-time coupling by putting access rules on layers. The access rule with the greatest impact was that for Xilinx code, only the bottom layer could directly include system header files. This made designing the bottom layer more difficult because its exported files also could not include system header files, otherwise the system header files would be included indirectly by unknowing clients and insulation would be lost. We already had a set of system utilities that supported most of the system functions that might be needed. By making this rule we could speed up compilations and decouple the majority of our software from operating system quirks.

Encapsulation in our definition also includes keeping the global name space clean. In the absence of namespace support on every platform, every exported symbol must be unique across the system. This requires the creation of a system wide naming convention that can be uniformly applied. This also requires restrictions on the use of certain compiler features, like #define, that can potentially cause conflicts between subsystems. We looked for a tool that could be smart enough to help automate this clean-up process, but could not find one. In the end, we decided that much of what could be done in this area was impractical within the time and resource constraints of the initial re-architecture into subsystems and layers.

Consequently, although we did create a system wide naming convention that applied to all new code, we grandfathered existing code. This can lead to inconsistent header files that contain classes that follow different naming schemes. To compensate, we allowed the use of typedefs to make all classes within a subsystem consistent with the naming convention, but did not require clients of the existing interfaces to change. We determined that certain naming conventions had to be followed to avoid run time errors and required that these be followed, but other than that made few changes to the existing names of classes.

Despite these few exceptions left until later releases, we expect the introduction of subsystems and all they imply to lead to substantial improvements in autonomy.

5.3.2 Sharing

The introduction of subsystems and layers does little to effect code sharing. The base/PM relationship that implements sharing in the manner of domain engineering[16] is still supported by the new architecture. Now however, the base and its PM code are contained within a single subsystem.

The creation of the generic algorithms layer is aimed directly at improving the other kind of sharing, sharing of smaller more general purpose code. In this regard, the architecture changes are expected to lead to significantly more code sharing of these generic algorithms.

5.3.3 Testing

In the previous Xilinx software architecture, the separation of PM code from base code made it difficult to independently test the software. This is because the PM created a dynamically loaded library that was difficult to use in the absence of the base code. The concept of subsystems allows us to consolidate more of the code together and make the subsystem integrator responsible for maintaining and running tests. By having a single point of contact for a large set of code it was felt we had a better chance of getting a solid aliveness testing methodology in place.

The notion of levelizable software is also directly addressed by the architectural changes. Recall that if the compile-time dependency graph of a system contains no loops, it is said to be levelizable[9]. By organizing the subsystems into layers and strictly defining the access rules for layers, the system is likely to be levelizable. With the additional rule that dependencies between subsystems within a given layer cannot cause a dependency loop, we can guarantee that the entire system is levelizable.

By factoring the system into layers, we get the additional benefit that we can build and test the lowest layer first and then on up the dependency graph.

Consequently, the architectural changes lead to testing improvements in four areas: allowing the base and PM code to be tested together, making the subsystem integrator responsible for all subsystem testing, making it easy to guarantee a levelizable system, and allowing the system to be tested layer by layer.

5.3.4 Comprehensibility

A final contribution of repackaging the software into subsystems is the ability to provide a consistent and easy to use mechanism to learn about and understand our software. As our software grew we found it increasingly difficult to prepare developers to write new code. The learning curve was steep due to the lack of accurate internal documentation. Clearly more documentation was required, but large detailed documents are often imposing to new developers and poorly maintain by the author, making their value dubious at best. What was needed was a simpler approach.

To this end, each subsystem was required to provide a *subsystem definition document*. This document is created and maintained by the subsystem integrator in a standard HTML format. The document is not comprehensive because it does not deal with subsystem implementation. Instead, it briefly describes the purpose of the subsystem, the files and libraries it produced and its exported interface. Further documentation of the exported interface is a set of HTML documents that are

generated directly from the exported interface header files using ccdoc[14]. In this manner a new or existing client of a subsystem can find an overview of the subsystem in the subsystem definition document, and detailed interface information in the header documentation.

We did not require that the subsystem implementation be described in an exported document, both because such a document would quickly become out of date and because it was "proprietary" knowledge not required by a subsystem's clients. To engineers who typically looked at a function's implementation before deciding whether it was what they wanted, this level of encapsulation and documentation was a revolutionary concept.

5.4 The Evolutionary Plan

After reaching consensus across the organization that the architecture would be reorganized into subsystems and layers, the final step was to schedule each of the packages to be converted into its corresponding subsystem. This task was made significantly more complex by the need to perform feature and device support work in the same time frame. In the past, Xilinx had tried to undertake major restructuring of its software, only to either fail or to wait many months before anything worked again. With the tight constraints of the release, neither of these risks was acceptable. Consequently we decided to convert the packages into subsystems in *waves*, changing only a fraction of the packages at a time. The exit criterion for each wave was defined to be working software that passed our internal engineering system test suite. The advantage of this process was that we would have working software after each wave. The disadvantage was that engineers had to create a different software environment for each wave, each with a different mix of old (packages) and new (subsystems). In order to mitigate risk, the plan introduced inefficiency by requiring almost everything in the system to change for every wave.

To prepare for each wave, the new subsystems were created several weeks in advance of the wave in which they were first used. This provided each subsystem with a trial period where it could be used locally but was not required for a wave to complete. To aid this mechanism we instituted a nightly build process where all the software released that day was built that night. These nightly builds gave us the chance to release a subsystem in one wave and then attempt to use it without affecting all existing clients in a full build.

In this section we discussed the changes to the software architecture and how those changes reflected the key concepts and goals for the re-architecture effort. We began by evaluating the current architecture against the goals and key concepts. We then chose the key concepts of autonomy, sharing, testing, and comprehensibility to be given first priority in the redesign effort. Based on the highest priority key concepts we proposed several different architectural solutions intended to address them and collected the best features of these proposals into a coherent document called the *system architecture vision*. We selected two of the ideas from the vision, deciding to create a generic algorithm layer and to re-factor the system into subsystems and layers. Finally, we created a plan to evolve from the starting point of our existing software architecture with limited risk. This plan called for the move from packages to subsystems to take place in several waves, ensuring that the system still functioned after each wave was complete.

In the following sections, we discuss the implementation of this plan and evaluate the effectiveness of our efforts to date.

6. Implementation

As of this writing, the initial transition to layers and subsystems is complete, and developers have been working with the new architecture for a few months. The wave plan succeeded initially, but after 3 waves, developers rebelled because the waves required a series of changes that needed to be re-visited for every wave. As a result, the final wave was more of a tidal wave that swept in all remaining changes. At that point, everyone understood the process well enough that we felt the risk involved in such a large change was justified by the time saved.

The major implementation hurdles to date have been more related to people and group dynamics than to technical issues. Initially, many engineers were somewhat confused as to what was actually happening, often because they were uninterested or too busy with other issues to take the time to really understand the process. Instead of riding the waves of change, unaware developers were hit by them, suddenly discovering that all their suppliers had new interfaces. Nevertheless, most of the work has been completed and we are not significantly behind schedule.

7. Results and Evaluation

As we said in Section 1, the re-architecture process is ongoing and will continue for many years as the software continues to change. However, we can begin to evaluate the initial two changes to the software architecture: creating a generic algorithms layer and re-factoring into subsystems and layers. These tasks are themselves incomplete, but preliminary results are encouraging. We detail our intermediate evaluation in terms of the key concepts these changes are intended to affect most.

7.1 Autonomy

At the time of this writing, it appears that there has been significant improvement in the ability of our engineers to work autonomously on their code. By separating the interface of a subsystem from its implementation and by placing controls on that interface, we have seen far fewer integration problems. Requiring engineers to

Table 1: Reduction in included files for a typical set of files

File (Level - see Fig. 2)	Num. Included Files Before	Num. Included Files After	Ratio: Before/ After
A (level 2)	44	6	7.3
B (level 3)	100	15	6.7
C (level 4)	217	97	2.2
D (level 5)	229	92	2.3
E (level 5)	327	223	1.5
F (level 5)	502	272	1.8
G (level 5)	266	110	2.4
H (level 5)	289	161	1.8
I (level 5)	321	156	2.1
Average			3.1

obtain approval for interface changes is cumbersome, but it makes engineers consider the impact of those changes.

In terms of insulation, results to date have been positive but depend greatly on the amount of time the engineer spent re-designing the interface to their subsystem. The first engineers to begin implementing their new fully insulated classes were greatly excited. After they spent weeks at a time working upon a single class and realizing that time was slipping away, the amount of insulation began to decrease dramatically. Engineers with less time often did almost nothing to redesign their interface. We have begun a detailed review of each interface to guide future work in this area.

In general, the engineers working on the lowest levels of the system (see Fig. 2) started working on their subsystems while other engineers were still working on the previous release. As a result, the most progress was made in the most heavily used portions of the system, which provides the most leverage to improve autonomy and reduce overall compilation times. This effect can be seen in Table 1, which shows the reduction in include file count for several files selected at random from various parts of the system. This is an important metric both because the number of included files is a rough measure of autonomy of a subsystem (more included files indicates less autonomy) and because including fewer files usually means faster compilations. These files were also selected because they are typical and because they have essentially the same functionality in the old system and the new. The levels in the table refer to the level numbering system shown in Fig. 2. Because of the additional work spent on insulating the lowest levels of the system, the most dramatic ratios of the numbers of included files can be seen for files in levels 2 and 3. For levels 4 and 5, more files are included

Table 2: Reduction in compilation times for modules that are essentially unchanged

Module	Compile Time (s) Before	Compile Time (s) After	Ratio: Before/ After
A (level 2)	16	13	1.2
B (level 3)	44	25	1.8
C (level 5)	110	90	1.2
D (level 5)	513	335	1.5
E (level 5)	585	396	1.5
F (level 5)	75	50	1.5
Average			1.5

because the code is at a higher level of abstraction. Moreover, most of the included functionality is from levels 2, 3, and 4, which have not been as well insulated. As a result, the ratio of the numbers of included files is not as dramatic.

The reduction in the number of included files is also reflected in compilation time of the system. However, compilation times are difficult to compare both because the functionality of any significant subset of the system has increased and because computer hardware and networks are continually being upgraded. However, despite increases in functionality, compilation time for the complete system has been reduced from approximately 22 hours to approximately 5 hours. (The compilation happens in parallel, but the times quoted are the sum of the times from each machine.) Consequently, despite any hardware improvements, it is safe to say that compilation time has decreased.

Improvements in compilation time can also be seen in Table 2, which shows compilation times for several modules performed under controlled conditions: on a 200 MHz UltraSparc machine running Solaris 2.6. These modules were selected because they have remained relatively unchanged. It is meaningless to compare portions of the system where most of the improvements were made because the structure and function of the code is dramatically different. We can only extrapolate from the overall compilation times to estimate that the largest compilation improvements are in the re-written portions of the system. However, even when the module is essentially unchanged, because the subsystems on which the module depends has been insulated, the compilation time has decreased.

Although insulation can improve autonomy, it can also adversely affect the runtime of the application. In one case, several in-line functions were re-introduced into an insulated subsystem because of the overhead of the function call. In each case, the function was called so many times that the function call overhead consumed 1-

2% of the overall runtime of an application that ran several hours. Faced with this runtime overhead, the in-line function was re-introduced. We plan to introduce alternative interfaces for clients with such special requirements.

As for encapsulation, the best we can say so far is that we have exposed our engineers to the idea of encapsulation. A few of the engineers did take this concept to heart and completely encapsulated their classes. These classes are now benefiting from this work in that they can have their class implementation changed without affecting their clients. Most engineers went into this project assuming that they were going to completely encapsulate every class. Once they started this process and saw the amount of time it took, they often retreated to insulation. This was acceptable because insulation provided immediately visible benefits that would encourage engineers to return to encapsulation as time permits.

The overall effect of the re-architecture effort on autonomy has been dramatic, particularly in the area of insulation. The use of encapsulation is more difficult to quantify, but we expect it will be more noticeable as the system continues to change.

7.2 Sharing

Improved sharing via the generic algorithm layer is also a long term investment. To date, several algorithms have been added to the layer and we have had at least one successful re-use. We expect greater utility over time, but unlike the optimistic predictions of early advocates of re-use, do not expect dramatic results from this effort.

7.3 Testing

Because we are not yet at that point in the process, we have not yet seen reductions in our testing burden or bug count. To provide a starting point for improved testing, there will be a special build, called a test build, used by developers to work on their internal tests. Each subsystem integrator will use this build to determine the amount of code coverage for their subsystem. Although we have tried to increase code coverage in the past, tight coupling with other developer's code was often used as an excuse for poor test coverage We believe that most engineers will be horrified by the lack of coverage and spend time increasing it. With this initial code coverage benchmark for the new software architecture, we can require increases in test coverage in future releases.

7.4 Comprehensibility

As with the other key concepts, comprehensibility is difficult to quantify and results have been mixed. In terms of documentation, we have created an on-line internal tools documentation area that contains the subsystem definition document for each subsystem. We have installed the ccdoc tool that creates documentation from C++ header files. Developers need only provide a specified type of comment in their interface header files and these will be added to the generated documentation. Even with all of this working, we hear anecdotally that not many people are using the browsing facilities. This is likely because most engineers have spent years

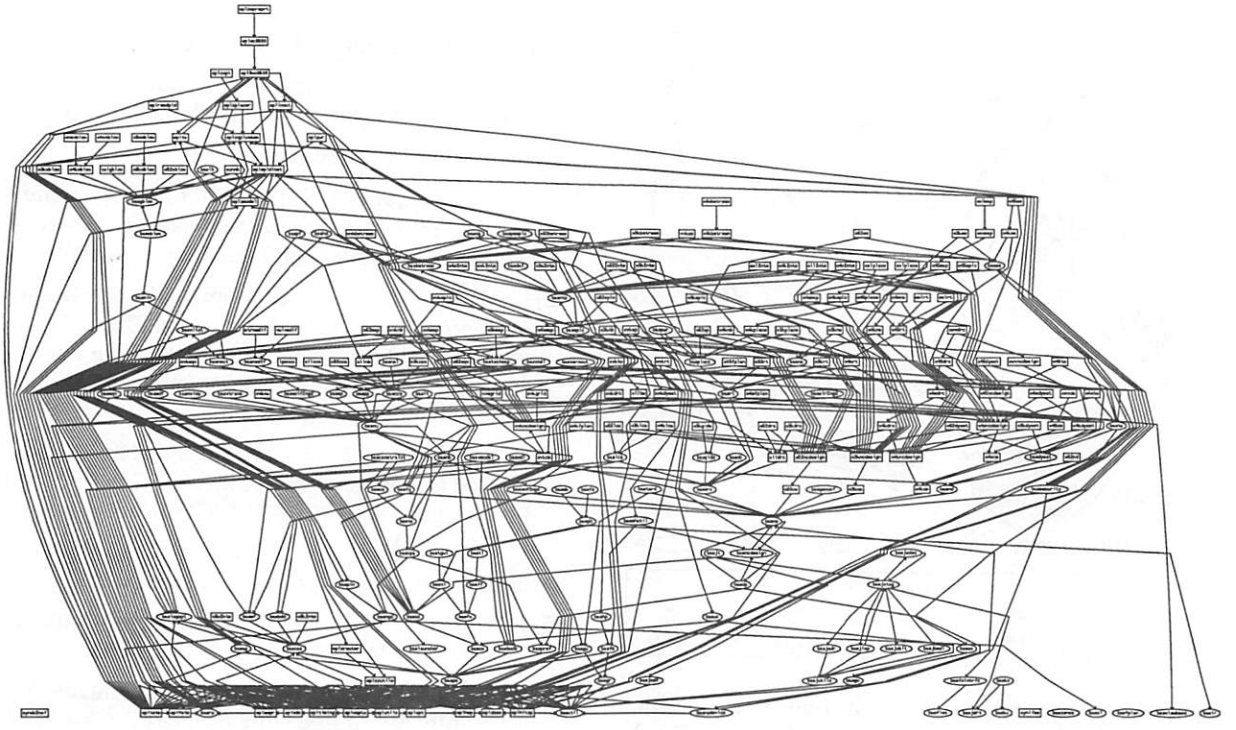


Figure 3. Topologically sorted graph of package include dependencies (before re-architecture).

reading header files directly and are still most comfortable doing so. Perhaps with the arrival of new engineers, this new browsing facility will become more useful. Similarly as we begin to move toward a heterogeneous mix of C++ and Java we might find these facilities more widely used.

One kind of comprehensibility where improvement is more readily apparent is the documentation of the overall architecture of the system. Drawings such as Fig. 2 provide a simple, high-level picture of the intended architecture. A complete drawing of all the subsystems and their include dependencies shown in Fig. 4 provide a more complete view of the system architecture. This graph is topologically sorted, which reveals that the compilation order of the system can be mapped back to the layers of Fig. 2. Note also that redundant edges (a direct edge to a subsystem included indirectly) have been removed. Fig. 4 also shows two remaining cyclic dependencies in the compilation order (upper right). These are in an isolated part of the system and will be eliminated soon.

Comparing Fig. 4 with Fig. 3, which was generated in the identical fashion from the previous architecture, it is quite apparent that the new architecture is easier to comprehend and work with at this level of abstraction.

8. Conclusions

In this paper, we introduced a six-step process by which Xilinx has made an initial iteration at re-architecting its software system. We have followed the progression through these steps and discussed the flow from analysis, to goals, to key concepts, to planning, to implementation and finally to evaluation. We have shown that we can modify a large legacy software system and create a software architecture that better balances among key concepts that reflect the demands of business, requirements of software engineering, and OOAD principles. Although this first iteration of our re-architecture is not yet complete, already we have seen significant gains in developer productivity.

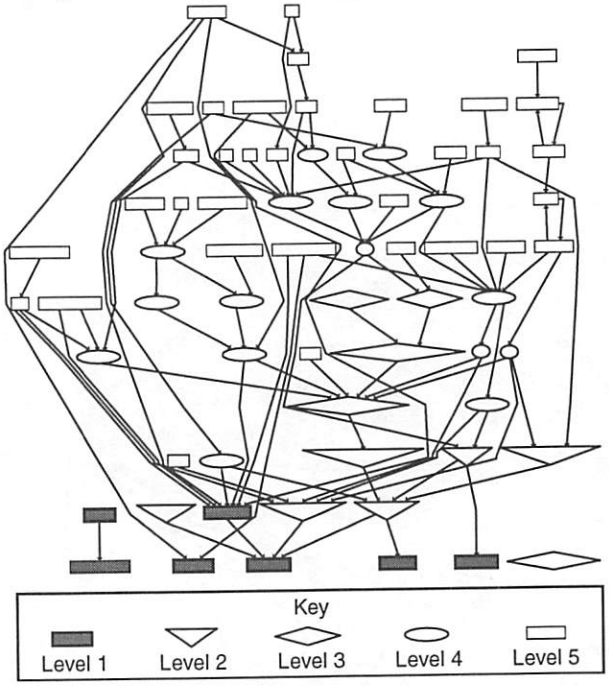


Figure 4. Topologically sorted graph of subsystem include dependencies (after re-architecture). Levels refer to Fig. 2.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] B. Foote and W. F. Opdyke, "Lifecycle and Refactoring Patterns that Support Evolution and Reuse," First Conference on Patterns Languages of Programs (PLoP '94). Monticello, Illinois, August 1994. *Pattern Languages of Program Design*, edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995

[3] G. Booch, *Object-Oriented Analysis and Design: with applications*. Benjamin/Cummings, 1994.

[4] S. McConnell, *Rapid development: taming wild software schedules*. Microsoft Press, 1996.

[5] http://www-cse.ucsd.edu/users/wgg/swevolution.html

[6] http://www.bell-labs.com/user/hpsiy/research/evolution.html

[7] http://www.comp.lancs.ac.uk/projects/RenaissanceWeb/

[8] http://www.sei.cmu.edu/reengineering/

[9] J. Lakos, "Large-Scale C++ Software Design," Addison-Wesley Professional Computing Series

[10] http://www.xilinx.com/company

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[12] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," Proceedings of the 29th ACM/IEEE conference on Design automation conference, 1992, Page 536.

[13] E. S. Ochotta, *et al*, "A Novel Predictable Segmented FPGA Routing Architecture," in FPGA '98, Proceedings of 1998 ACM/SIGDA intl. symp. on FPGAs, pp 3-11.

[14] http://www.joelinoff.com/ccdoc/index.html

[15] C.W. Krueger, "Software Reuse." ACM Computing Survey, vol. 24, no. 2, pp. 131-182, 1992.

[16] E. Mettala and M.H. Graham, "The Domain-Specific Software Architecture Program," CMU/SEI-92-SR-9, 1992.

Supporting Automatic Configuration of Component-Based Distributed Systems*

Fabio Kon† Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue, Urbana, IL 61801-2987 USA
{f-kon,roy}@cs.uiuc.edu
http://choices.cs.uiuc.edu

Abstract

Recent developments in Component technology enable the construction of complex software systems by assembling together off-the-shelf components. However, it is still difficult to develop efficient, reliable, and dynamically configurable component-based systems. Components are often developed by different groups with different methodologies. Unspecified dependencies and behavior lead to unexpected failures.

Component-based software systems must maintain explicit representations of inter-component dependence and component requirements. This provides a common ground for supporting fault-tolerance and automating dynamic configuration.

In this paper, we present a generic model for reifying dependencies in distributed component systems and discuss how it can be used to support automatic configuration. We describe our experience deploying the framework in a CORBA-compliant reflective ORB and discuss the use of this model in a new distributed operating system.

1 Introduction

Research on object-oriented technology and its intensive use by the industry has led to the development of *component-oriented programming*. Rather than being an alternative to object-orientation, component technology extends the initial concepts of objects. It stresses the desire for independent pieces of software that can be reused and combined in different ways to implement complex software systems.

Recently developed component architectures [Ham97, Den97, OMG97] support the construction of sophisticated systems by assembling together a collection of off-the-shelf software components with the help of visual tools or programmatic interfaces. However, there is still very little support for managing the interactions between components. Components are created by different programmers, often working in different groups with different methodologies. It is hard to create robust and efficient systems if the dynamic dependencies between components are not well understood. It is very common to find cases, in both legacy and component-based systems, in which a module fails to accomplish its goal because an unspecified dependency is not properly resolved. Sometimes, the graceful failure of one module is not properly detected by other modules leading to system failure.

A similar problem can be detected in a different context. Current systems are continuously being updated and modified. For example, system administrators working on UNIX or Windows NT environments must be aware of security announcements on a daily basis and be prepared to update the operating system kernel with security patches. In addition, users demand new versions of applications such as web browsers, text editors, software development tools, and the like. Often, building and installing a

*This research is supported by a grant from the National Science Foundation, NSF 98-70736.

†Fabio Kon is supported in part by a grant from CAPES, the Brazilian Research Agency, proc.#1405/95-2.

new package requires that a series of other tools be updated.

Users of workstations and personal computers are also not free from the burden of system or account maintenance. In environments like MS-Windows, the installation of some applications is partially automated by "wizard" interfaces which directs the user through the installation process. However, it is common to face situations in which the installation cannot complete or in which it completes but the software package does not run properly because some of its (unspecified) requirements are not met. In other cases, after installing a new version of a system component or a new tool, applications that used to work before the update, stop functioning. It is typical that applications on MS-Windows cannot be cleanly uninstalled. Often, after executing special uninstall procedures, "junk" libraries and files are left in the system.

The problem behind all these difficulties is the lack of a model for representing the dependencies among system and application components and mechanisms for managing these dependencies.

We argue that operating system and middleware environments must provide support for representing the dependencies among software components in an explicit way. This representation can then be manipulated in order to implement software components that are able to configure themselves and adapt to ever changing dynamic environments.

By reifying the interactions between system and application components, system software can recognize the need for reconfiguration to better support fault-tolerance, security, quality of service, and optimizations. In addition, it gains the means to carry out this reconfiguration without compromising system stability and reliability and with minimal impact in performance.

Our research builds on previous and ongoing work on software architecture [SG96], dynamic reconfiguration of distributed systems [HWP93, Hof94, SW98], and quality of service specification [FK98, LBS+98]. Our long-term goal is to develop a generic model for automatic configuration that can be applied to modern component architectures.

1.1 Paper Contents

The initial objective of our research is the support for representing dependencies among software components in an explicit way. With that support, we develop mechanisms that utilize this representation to perform automatic (re)configuration of software components in dynamic environments.

This paper describes our model for representing component prerequisites (section 2.1) and runtime inter-component dependence (section 2.2). Although we describe the implementation of a framework for reifying inter-component dependence, the details about the implementation of prerequisites are out of the scope of this paper and will be addressed in a future document.

Section 3 presents two application scenarios: section 3.1 describes our experience using the framework to support on-the-fly reconfiguration of *dynamicTAO*, a reflective CORBA-compliant ORB and section 3.2 discusses the use of our model in the *2K* distributed operating system.

After discussing related work in section 4, we describe our plans for the future in section 5 and present our conclusions in section 6.

2 Inter-Component Dependence

To address the problems described in the previous section, a configuration system must explore two distinct kinds of dependencies:

1. Requirements for loading an inert component into the runtime system (called *prerequisites*).
2. *Dynamic dependencies* among loaded components in a running system.

As long as the system knows exactly what the requirements are for installing and running a software component, the installation and configuration of new components can be automated. As a byproduct of this knowledge, component performance can be improved by analyzing the dynamic state of system resources, analyzing the characteristics of each component, and by configuring them in the most efficient way.

Also, if the system knows what the dynamic dependencies among running components are, it can (1) better handle exceptional behavior that could potentially trouble component operation, and (2) support dynamic reconfigurations of large systems by replacing individual components on-the-fly.

Prerequisites and runtime dependencies are two distinct forms of the same entity. Prerequisites usually are expressed as dependencies on "persistent" hardware and software components while runtime dependencies refer to dynamic, possibly volatile, components. In particular, if one freezes a component's state (including its runtime dependencies) and stops it, one could later resume its execution by using the frozen runtime dependencies as the prerequisites for reloading the component. However, in order to make the model as clear as possible, we are going to treat prerequisites and runtime dependencies as separate entities. Prerequisites usually refer to hardware resources, QoS requirements, and software services. Runtime dependencies refer to loaded software components. Thus, we believe that the separation of concepts is justifiable. In the future, after the basic problems are solved, we may consider to unify these concepts in order to build a simpler and more generic model.

2.1 Prerequisites

The prerequisites for a particular inert component must specify any special requirement for properly loading, configuring, and executing that component. We consider three different kinds of information that can be contained in a list of prerequisites.

1. The nature of the hardware resources the component needs.
2. The capacity of the hardware resources it needs.
3. The software services (i.e., components) it requires.

The first two items may be used by a distributed Resource Management Service to determine where, how, and when to execute the component. QoS-aware systems can use these data to enable proper admission control, resource negotiation, and resource reservation. The last item is the one which determines which auxiliary components must be loaded and in which kind of software environment they will execute.

The first two items can be expressed by QoS specification languages [FK98, LBS+98]. The third item is equivalent to the *requires* clause in module interconnection languages like, for instance, the one used in Polylith [Pur94]. We are in the process of analyzing existing specification languages to study which ones would best fit our needs. The language must allow processing specifications at execution time with little overhead. We will deploy initial prototypes in *2K*, a new CORBA-based distributed operating system [KSC+98, CNM98] currently under development. The main purpose of this paper, however, is to describe the design and implementation of the infrastructure for representing runtime dependencies presented next.

2.2 Dynamic Dependencies

In our model, each component is managed by a *component configurator* which is responsible for storing the dependencies between a specific component and other system and application components.

Depending on the way it is implemented, a component configurator may be able to refer to components running on a single address space, on different address spaces and processes, or even running on different machines in a distributed system. Figure 1 depicts the dependencies that a component configurator reifies.

Each component C has a set of *hooks* to which other components can be attached. These are the components on which C depends and are called *hooked components*. There might be other components that depend on C, these are called *clients*. In general, each time one defines that a component C_1 depends on a component C_2, the system should perform two actions:

1. attach C_2 to one of the hooks in C_1 and
2. add C_1 to the list of clients of C_2.

As an example, consider a web browser that specifies, in its list of prerequisites, that it requires a TCP/IP service, a window manager, and a local file

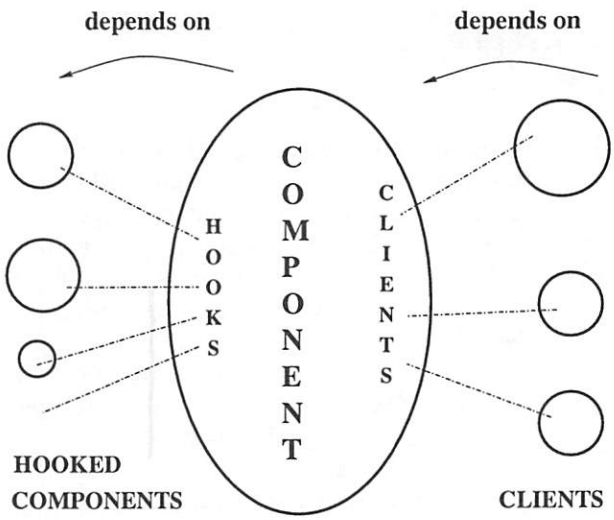


Figure 1: Reification of component dependence.

service. Its component configurator should maintain a hook for each of these services. When the browser is loaded, the system must verify whether these services are available in the local environment. If they are not, it must create new instances of them. In any case, references to the services are stored in the browser configurator hooks and may be later retrieved and updated if necessary.

2.2.1 The ComponentConfigurator class

The reification of runtime dependencies is accomplished by assigning one *ComponentConfigurator* object to each component. A simplified declaration of the *ComponentConfigurator* class in pseudo-C++ follows. Figure 2 shows a schematic representation of some of its method calls.

The class constructor receives a pointer to the component implementation as a parameter. It can be later obtained through the *implementation()* method.

The *hook()* method is used to specify that this component depends upon another component and *unhook()* breaks this dependence. The *registerClient()* and *unregisterClient()* methods are similar to *hook()* and *unhook()* but they specify that other components (called clients) depend upon this component.

```
class ComponentConfigurator {
public:
 ComponentConfigurator(Object *implementation);
 ~ComponentConfigurator ();

 int hook (const char *hookName,
           ComponentConfigurator *component);
 int unhook (const char *hookName);
 int registerClient
      (ComponentConfigurator *client,
       const char *hookNameInClient = NULL);
 int unregisterClient
      (ComponentConfigurator *client);

 int eventOnHookedComponent
      (ComponentConfigurator *hookedComponent,
       Event e);
 int eventOnClient
      (ComponentConfigurator *client,
       Event e);

 char *name ();
 char *info ();
 DependencyList *listHooks ();
 DependencyList *listClients ();
 ComponentConfigurator *
   getHookedComponent (const char *hookName);

 Object *implementation ();
}
```

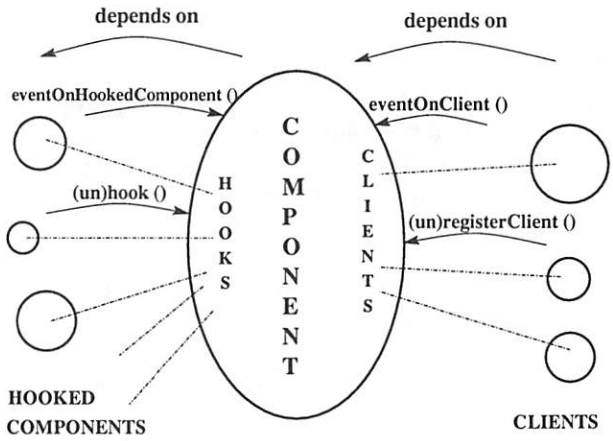


Figure 2: Methods for specifying dependencies and sending events.

eventOnHookedComponent() announces that a component which is attached to this component has generated an event. The *ComponentConfigurator()* class is subclassed to implement different behaviors when events are reported. Examples of common events are the destruction of a hooked component, the in-

ternal reconfiguration of a hooked component, or the replacement of the implementation of a hooked component.

eventOnClient() is similar to the previous method but it announces that a client has generated an event. This can be used, for example, to trigger reconfigurations in a component to adapt to new conditions in its clients. Our reference implementation defines a basic set of events including *DELETED*, *FAILED*, *RECONFIGURED*, *REPLACED*, and *MIGRATED*. Applications can extend this set by defining their own events.

name() returns a pointer to a string containing the name of the component and *info()* returns a pointer to a string containing a description of the component. Specific *info()* implementations can return different kinds of information like a list of configuration options accepted by the component, or a URL for its documentation and source code.

listHooks() returns a pointer to a list of *DependencySpecifications*. A *DependencySpecification* is a structure defined as

```
struct DependencySpecification {
 const char *hookName;
 ComponentConfigurator *component;
};
```

listClients() returns a pointer to a list of *DependencySpecifications* corresponding to the components that depend on this component (its clients) and the name of the hooks (in the client's ComponentConfigurator) to which this component is attached.

Finally, *getHookedComponent()* returns a pointer to the configurator of the component that is attached to a given hook.

2.2.2 Towards Automatic Configuration

As discussed above, reified inter-component dependence can help the automation of configuration processes. By scanning the list of prerequisites, the operating system or middleware can be certain that all hardware and software requirements for the execution of a particular component are met before it is initiated. This can avoid a large number of problems that are common in existing systems where the lack of a particular component or resource is only detected after the application is running.

The dynamic dependence information, in its turn, enables the reconfiguration of components that are already running. In addition, it provides important information for implementing fault-tolerance and smooth exception handling in an environment of centralized or distributed components.

As an example, consider the deletion of a component containing our *ComponentConfigurator* class. Different policies for dealing with component deletion can be adopted. In general, when a component C is destroyed, an announcement must be made to components that depend on C and to components on which C depends. The following piece of pseudo-C++ code illustrates this process with a conservative implementation of the *ComponentConfigurator* destructor.

```
ComponentConfigurator::~ComponentConfigurator()
{
 for (c in hookedComponents) {
  c.configurator->unregisterClient (this);
 }

 for (c in clients) {
  c.configurator->
     eventOnHookedComponent (this,
                             DELETED);
 }

 // delete list of hooks and hookedComponents
 // delete list of clients
 // release resources
 // delete component implementation
}// ~ComponentConfigurator ()
```

Implementations of this destructor can be specialized to adjust its behavior to different component types and to meet special requirements. Also, different component types must implement methods such as *eventOnHookedComponent()* in proper ways to take care of the different kinds of dependencies. In an extreme case, deleting a component will cause all components that depend on it to be deleted. In the other extreme case, these other components will only be notified and nothing else will change. In most of the cases, we expect that these components will try to reconfigure themselves in order to deal with the loss of one of its dependencies.

The problem with this implementation is that the complete destruction of the component only takes

place if all the method calls to hooked components and clients return. If any of these calls block, the component is not deleted. This problem is particularly important if some of the clients decide to initiate their own destruction as a result of the call to *eventOnHookedComponent()* and a long chain of calls is established.

A naïve solution to this problem could be to execute the method calls asynchronously, for example, by creating new threads to perform the calls. This solution would incur in the additional cost of creating new threads and could lead to dangerous situations as a C++ component could try to call a method on another component after the latter is destroyed.

Thus, it seems that we are trapped between a safe, conservative solution that might block indefinitely and a liberal but unsafe solution that may crash the whole system by executing invalid code. We have been studying this problem and, in [KC98], we discuss solutions that lie somewhere between these two extremes. They are as safe as the conservative one but are less subject to blocking.

2.2.3 Managing Dependencies

The use of our model in a language like C++ requires strict collaboration from the component developer to conform to proposed guidelines. It is also important that all the communication between components be done through controlled interfaces. In order to avoid a proliferation of programming errors related to dependence reification, it would be necessary to develop special languages, compilers, and runtime systems to guarantee the safety of component execution and reconfiguration.

A cleaner solution would be to use existing reflective languages and environments. Iguana [GC96] and OpenC++ [Chi95], for example, are extensions to C++ that reify several features of this language, allowing dynamic modification of their implementations. In these languages, it would be possible to instrument method invocation to take care of dependence maintenance.

However, a major goal of our research is not to limit the implementation to a particular programming language and only use widely accepted standards. We could also tie together the mechanisms for communication and dependence representation using, for example, abstract connectors [SDZ96]. But this could limit the expressiveness of the model. Our objective is to develop a generic methodology that could be utilized in a large number of heterogeneous environments. These requirements can only be met by using a standard architecture like CORBA.

2.3 CORBA ComponentConfigurator

CORBA permits the integration of components written in different programming languages on heterogeneous environments. In addition, CORBA's (remote) method invocation mechanism can be decoupled from the base language method call. Thus, it is possible to guarantee that bad CORBA references are not translated into bad base language references (like dangling C++ pointers for example). Instead, exceptions are neatly handled by the runtime and the application is informed of its occurrence.

In the CORBA implementation of our model, a *DependencySpecification* stores a CORBA Interoperable Object Reference (IOR) so that the *ComponentConfigurator* is able to reify dependencies among distributed components. Prerequisites for software components can be specified either in terms of persistent IORs [Hen98] or in terms of service type and attributes. In the former case, an implementation repository can be used to dynamically create a new CORBA object if one is not available. In the latter case, the CORBA Trading Object Service [OMG98] can be used to locate an instance of the server component that meets the requirements specified by the given attributes.

When a CORBA component is destroyed, the component implementation (or the ORB) must call the configurator destructor so that it can tell its clients that the destruction is taking place. If a node crashes or if the whole process containing both the component and the configurator crash, it might not be possible to execute the configurator destructor. In this case, the clients will not be informed of the component destruction. Subsequent CORBA invocations to the crashed component will raise an exception announcing that the object is not reachable or that it does not exist. In this case, it is the responsibility of the client component to locate a new server component and update its *ComponentConfigurator*.

As future work, we intend to perform experiments with the different ways of using the CORBA *ComponentConfigurator* to manage distributed applications. In particular, component configurators can be (1) co-located with their respective component implementations, (2) located in a separate process in the same machine or (3) located in a centralized node on the network while the component implementations are distributed. We will investigate the benefits of the different approaches.

2.4 Implementation Status

We have implemented prototypes of the *ComponentConfigurator* for centralized applications in C++ and Java. The C++ implementation was deployed in the *dynamicTAO* ORB as described in section 3.1. We have recently completed an implementation of distributed *ComponentConfigurator*s based on CORBA.

We plan to extend the Java implementation to support Java Bean components and distributed object communication with Java RMI. We will, then, work on the interoperability among different implementations of the model in different component architectures.

3 Application Scenarios

This section describes the deployment of the *ComponentConfigurator* framework in *dynamicTAO*, a reflective Object Request Broker. It illustrates how our model can be used to represent and manipulate the internal structure of a legacy system, enabling dynamic reconfiguration. We, then, discuss how this framework will be used to support architectural awareness in the *2K* distributed operating system.

3.1 *dynamicTAO*

One of the major constituent elements of *2K*, a distributed operating system our group is developing [KSC+98, CNM98], is a reflective middleware layer based on CORBA. After carefully studying existing Object Request Brokers, we came to the conclusion that the TAO ORB [SC99] would be the best starting point for developing our infrastructure. TAO is a portable, flexible, extensible, and configurable ORB based on object-oriented design patterns. It uses the *Strategy* design pattern [GHJV95] to separate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded.

TAO is primarily targeted for static hard real-time applications such as Avionics systems [HLS97]. Thus, it assumes that, once the ORB is initially configured, its strategies will remain in place until it completes its execution. There is very little support for on-the-fly reconfiguration.

The *2K* project seeks to build a flexible infrastructure to support adaptive applications running on dynamic environments. On-the-fly adaptation is extremely important for a wide range of applications including the ones dealing with multimedia, mobile computers, and dynamically changing environments.

The design of *2K* depends on *dynamicTAO*, an extension of TAO that enables on-the-fly reconfiguration of its strategies. *dynamicTAO* exports an interface for loading and unloading modules into the ORB runtime, and for inspecting the ORB configuration state. The architecture can also be used for dynamic reconfiguration of servants running on top of the ORB and even for reconfiguring non-CORBA applications.

3.1.1 Problems Encountered

Reconfiguring a running ORB while it is servicing client requests is a difficult task that requires careful consideration. There are two major classes of problems.

Consider the case in which *dynamicTAO* receives a request for replacing one of its strategies (S_{old}) by a new strategy (S_{new}). The first problem is that, since TAO strategies are implemented as C++ objects that communicate through method invocations, before unloading S_{old}, the system must be sure that no one is running S_{old} code and that no one is ex-

pecting to run S_{old} code in the future. Otherwise, the system could crash. Thus, it is important to assure that S_{old} is only unloaded after the system can guarantee that its code will not be called.

The second problem is that some strategies need to keep state information. When a strategy S_{old} is being replaced by S_{new}, part of S_{old}'s internal state may need to be transfered to S_{new}.

These problems can be addressed with the help of the *ComponentConfigurator* which is used to reify the dependencies among strategies, instances of *dynamicTAO*, and servants.

3.1.2 DomainConfigurator and TAOConfigurator

Each process running the *dynamicTAO* ORB contains a *ComponentConfigurator* instance called *DomainConfigurator*. It is responsible for maintaining references to instances of the ORB and to servants running in that process. In addition, each instance of the ORB contains a customized subclass of *ComponentConfigurator* called *TAOConfigurator*.

TAOConfigurator contains hooks to which *dynamicTAO* strategies are attached. A *NetworkBroker* implements a simple TCP-based protocol that allows remote entities to connect to the process to inspect and change the configuration of *dynamicTAO* by loading new strategies and attaching them to specific hooks. Local servants and remote CORBA clients can also access the *Configurator* objects through a programmatic CORBA interface. Figure 3 illustrates this mechanism when a single instance of the ORB is present.

If necessary, individual strategies may have their own customized subclass of *ComponentConfigurator* to manage their dependencies upon ORB instances and other strategies. These subclasses may also store references to client connections that depend on them. With this information, it is possible to decide when a strategy can be safely unloaded.

Consider, for example, the three concurrency strategies supported by *dynamicTAO*: Single-Threaded Reactive [Sch94], Thread-Per-Connection, and Thread-Pool. If the user switches from the Reactive or Thread-Per-Connection strategies to any other concurrency strategy, nothing special needs to be done. *dynamicTAO* may simply load the new strategy, update the proper *TAOConfigurator* hook, unload the old strategy, and continue. Old client connections will complete with the concurrency policy dictated by the old strategy. New connections will utilize the new policy.

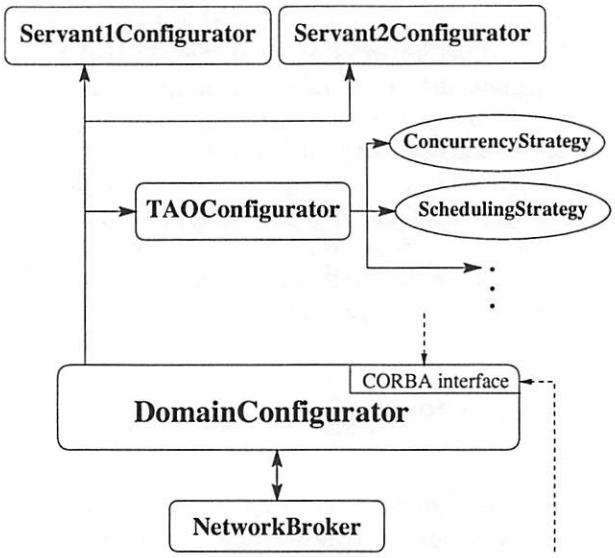


Figure 3: Remote Configuration of *dynamicTAO* strategies.

However, if one switches from the Thread-Pool strategy to another one, special care must be taken. The Thread-Pool strategy we developed maintains a pool of threads that is created when the strategy is initialized. The threads are shared by all incoming connections to achieve a good level of concurrency without having the runtime overhead of creating new threads. A problem arises when one switches from this strategy to another strategy: the code of the strategy being replaced cannot be immediately unloaded. This happens because, since the threads are reused, they return to the Thread-Pool strategy code each time a connection finishes. This problem can be solved by a *ThreadPoolConfigurator* keeping information about which threads are handling client connections and destroying them as the connections are closed. When the last thread is destroyed the Thread-Pool strategy signalizes that it can be unloaded.

Another problem occurs when one replaces the Thread-Pool strategy by a new one. There may be several incoming connections enqueued in the strategy waiting for a thread to execute them. The so-

lution is to use the *Memento* pattern [GHJV95] to encapsulate the old strategy state in an object that is passed to the new strategy. An object is used to encapsulate the queue of waiting connections. The system simply passes this object to the new strategy which then takes care of the enqueued connections.

Our group is currently expanding the set of *dynamicTAO* strategies that can be replaced on-the-fly. The *TAOConfigurator* will have hooks for holding strategies for connection management, concurrency, (de)marshalling, request demultiplexing, method dispatching, scheduling, and security. An explicit knowledge of the dependencies among the ORB components is essential for implementing dynamic reconfiguration safely.

3.2 Architectural Awareness in *2K*

In contrast to existing systems where a large number of non-utilized modules are carried along with the basic system installation, the *2K* operating system is based upon a "what you need is what you get" (WYNIWYG) model. The system configures itself automatically and loads the minimum set of components required for executing user applications in the most efficient way. Components are downloaded from the network and only a small subset of system services are needed to bootstrap a node.

This is achieved by reifying the hardware and software prerequisites for each loadable component. As mentioned in section 2.1, the operating system can use this information to make sure that all the basic services that a component requires are available before the component is loaded. In addition, a distributed resource manager uses the specifications of the component hardware requirements to decide in which machine the component should be loaded and perform admission control and resource reservation. That way, one will not face a situation in which a component fails to execute its task with the desired quality of service because an unspecified dependency was not resolved.

As a component is loaded into the system, its prerequisites are scanned and all the specified services are made available. During this process, the system can incrementally build a dynamic graph of dependencies using the *ComponentConfigurator* framework.

The design of *2K* supports fault-tolerant, self-adapting systems by monitoring the environment and maintaining a representation of the dynamic structure of its services and applications. The CORBA implementation of the *ComponentConfigurator* framework reifies the distributed system dynamic structure.

When a *2K* component fails, the system inspects its dependencies and informs the proper components about the failure. The system may alternatively recover from a failure by replacing the faulty component with a new one. The same mechanism can be used for adapting the system and its components to changing parameters such as network bandwidth, CPU load, resource availability, user access patterns, etc.

4 Related Work

The idea of using prerequisites to represent the dependencies among operating system objects was introduced in the SOS operating system [SGH+89] developed at INRIA, France. In the SOS model, objects contain a list of *prerequisites* that must be satisfied before they are activated. Even though the idea was promising, it was not fully explored in that project. Prerequisites were only used to express that an object depends on the code implementing it. Not much experimentation was carried out [SGM89, Sha98]. SOS does not include a model for dynamic management of inter-component dependence.

Previous research in microkernels and customizable operating systems – such as Mach [Lop91], SPIN [BSP+95], Exokernel [KEG+97], and μChoices [LTC96] – developed low-level techniques for dynamic loading new modules to the operating system both in kernel and user space. Nevertheless, a high-level model for operating system reconfiguration is still inexistent. These previous works have not addressed a number of problems related to fault-tolerance and dynamic reconfiguration. Using the *ComponentConfigurator* framework, our research investigate answers to the following questions.

- What are the consequences of reconfiguring the operating system?
- When a system module is replaced, which other

modules are affected?

- How must those other modules react?
- When (re)configuring the system, which components must be loaded to meet the service demand and the required quality of service?
- If a system component fails, how can the system detect it and recover gracefully?

We are currently investigating languages for prerequisite specification. They must be able to represent hardware and quality of service requirements as well as dependencies on other software components. Thus, we believe that an ideal language for prerequisite specification will build on previous work on Architecture Description Languages [Cle96] and QoS Specification Languages [FK98, LBS+98].

Connector-based systems like UniCon [SDZ96] and software buses like POLYLITH [Pur94] separate issues concerning component functional behavior from component interaction. Our model goes one step further by separating inter-component communication from inter-component dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Our framework is independent of the paradigm for inter-component communication; it can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, etc.

Communication and dependence are often intimately related. But, in many cases, the distinction between inter-component dependence and inter-component communication is beneficial. For example, the quality of service provided by a multimedia application is greatly influenced by the mechanisms utilized by underlying services such as virtual memory, scheduling, and memory allocation (through the `new` operator). The interaction between the application and these services is often implicit, i.e., no direct communication (e.g. library or system calls) takes place. Yet, if the system infrastructure allows developers to establish and manipulate dependence relationships between the application and these services, the application can be informed of substantial changes in the state and configuration of the services that may affect its performance.

Differently from previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. As shown in section 3.1, the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

We are particularly interested in investigating the possibilities of applying results from previous and ongoing work in dynamic reconfiguration [HWP93, SW98, BBB+98] to standard architectures such as CORBA and Java Beans.

5 Ongoing and Future Work

The current implementation of the framework in C++ is being used in *dynamicTAO* as its dynamic reconfigurability is enhanced. In addition, the Java implementation is being used by researchers at the University of São Paulo to prototype a domain decomposition manager. This manager has two demonstration applications: a Distributed Information System for Mobile Agents [SGE98] and the parallelization of an Atmospheric Modeling System [Bar98].

Work on implementations of the framework in Java RMI is underway. As discussed in 3.2, the CORBA implementation of the *ComponentConfigurator* will be used in the *2K* operating system to support runtime architectural awareness as the basis for implementing fault-tolerant reconfigurable systems. The prerequisites model will be used for QoS-aware resource management. This will provide components with all the hardware and software resources they need to execute with the desired quality of service.

6 Conclusions

We have presented a model for runtime architectural awareness in centralized and distributed component-based systems. We believe that the reification of inter-component dependence and component prerequisites is fundamental for systems supporting fault-tolerant, reconfigurable components.

The model has been prototyped in Java, C++, and CORBA. The C++ framework was successfully deployed in *dynamicTAO*, a legacy system, which was

made aware of its own internal structure.

Future work in the *2K* operating system will demonstrate how the model behaves in a complex, distributed CORBA-based system.

Acknowledgments

We gratefully acknowledge the help provided by Manuel Román on the implementation of dynamicTAO. We thank Dilma Menezes, Francisco Ballesteros, and the members of the *2K* team for their feedback on the ideas presented in this paper. Finally, we thank the anonymous reviewers who contributed with valuable comments.

Availability

The framework source code in C++ and Java is available at `http://choices.cs.uiuc.edu/2k/ComponentConfigurator` .

The source code and detailed documentation for *dynamicTAO* can be found at `http://choices.cs.uiuc.edu/2k/dynamicTAO` .

References

[Bar98] Saulo Barros. The Regional Atmospheric Modeling System (RAMS) Project. `http://www.ime.usp.br/~rams/`, 1998.

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, UK, September 1998.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chabers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the OOPSLA '95*, pages 285–299, October 1995.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proceedings of The Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

[CNM98] Roy H. Campbell, Klara Nahrstedt, and M. Dennis Mickunas. 2K: A Component-Based Network-Centric Operating System. Project home page: `http://choices.cs.uiuc.edu/2K`, 1998.

[Den97] Adam Denning. *ActiveX Controls Inside Out.* Microsoft Press, Redmond, second edition, 1997.

[FK98] Svend Frølund and Jari Koistinen. Quality of Service Specification in Distributed Object Systems Design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[GC96] Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The iguana approach. In *Proceedings of Reflection '96*, pages 137–152, San Francisco, USA, April 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software.* Addison-Wesley, 1995.

[Ham97] Graham Hamilton. *JavaBeans specification.* Sun Microsystems, 1997. Available at `http://java.sun.com/beans/docs`.

[Hen98] Michi Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM*, 41(10), October 1998.

[HLS97] Tim Harrison, David Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Object Event Service. In *Proceedings of OOPSLA '97*, Atlanta, Georgia, October 1997.

[Hof94] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications.*

PhD thesis, University of Maryland, Department of Computer Science, January 1994. Technical Report CS-TR-3210.

[HWP93] Christine Hofmeister, E. White, and James M. Purtilo. SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications. *IEE Software Engineering Journal*, 8(2):95–101, March 1993.

[KC98] Fabio Kon and Roy H. Campbell. On the Role of Inter-Component Dependence in Supporting Automatic Reconfiguration. Technical Report UIUCDCS-R-98-2080, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1998.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Tom Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Saint Malo, France, October 1997. ACM.

[KSC+98] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[LBS+98] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS Aspect Languages and Their Runtime Integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, May 1998. To appear in Lecture Notes in Computer Science, Springer-Verlag.

[Lop91] Keith Lopere. Mach 3 kernel principles. *Open Software Foundation*, 1991.

[LTC96] W. S. Liao, S. Tan, and R. H. Campbell. Fine-grained, Dynamic User Customization of Operating Systems. In *Proceedings Fifth International Workshop on Object-Orientation in Operating Systems*, pages 62–66, Seattle, Washington USA, October 1996.

[OMG97] OMG. *CORBA Component Model RFP.* Object Management Group, Framingham, MA, 1997. OMG Document 97-05-22.

[OMG98] OMG. *CORBAservices: Common Object Services Specification.* Object Management Group, Framingham, MA, 1998. OMG Document 98-07-05.

[Pur94] James Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[SC99] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 1999. (to appear), available at `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.

[Sch94] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[SDZ96] Mary Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, May 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SGE98] Dilma Menezes da Silva, Marco Dimas Gubitoso, and Markus Endler. Sistemas de Informação Distribuídos para Agentes Móveis. In *Proceedings of the XXV Integrated Seminars in Software and Hardware (SEMISH'98)*, pages 125–140, Belo Horizonte, Brazil, August 1998. SBC. Available at `http://www.ime.usp.br/~dilma/papers/semish98.ps`.

[SGH+89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin,

and Cline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In Stephen Cook, editor, *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[Sha98] Marc Shapiro. Personal communication, July 1998.

[SW98] S. K. Shrivastava and S. M. Wheater. Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Application. In *Proceeding of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, May 1998.

Automating Three Modes of Evolution for Object-Oriented Software Architectures

Lance Tokuda, Don Batory

Department of Computer Science
University of Texas at Austin
Austin, TX 78712-1188
{unicron, dsb}@cs.utexas.edu

Abstract[1]

Architectural evolution is a costly yet unavoidable consequence of a successful application. One method for reducing cost is to automate aspects of the evolutionary cycle when possible. Three kinds of architectural evolution in object-oriented systems are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. This paper shows that all three can be viewed as transformations applied to an evolving design. Further, the transformations are automatable with refactorings — behavior-preserving program transformations. A comprehensive list of refactorings used to evolve large applications is provided and an analysis of supported schema transformations, design patterns, and hot-spot meta patterns is presented. Refactorings enable the evolution of architectures on an if-needed basis reducing unnecessary complexity and inefficiency.

1 Introduction

All successful software applications evolve [Par79]. During the 1970s, evolution and maintenance accounted for 35 to 40 percent of the software budget for an information systems organization. This number jumped to 60 percent in the 1980s. It was predicted that without a major change in approach, many companies will spend close to 80 percent of their software budget on maintenance [Pre92]. As applications evolve, so do their architectures. Architectures evolve for multiple reasons:

- **Capability** — to support new features or changes to existing features.
- **Reusability** — to carve out software artifacts for reuse in other applications.
- **Extensibility** — to provide for the addition of future extensions.
- **Maintainability** — to reduce the cost of software maintenance through restructuring.

We have observed that architectures also evolve for human reasons:

- **Experience.** Experienced employees can often design a better architecture based on their knowledge of the current architecture.
- **New Perspective.** New project members often have new ideas about how an architecture could or should be structured. Many organizations use a code ownership model which empowers new employees with the ability to realize their new designs.

While motivations vary, the methods used for evolving architectures appear to follow regular patterns, particularly for object-oriented applications. Three kinds of object-oriented architectural evolution are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. *Schema transformations* are drawn from object-oriented database schema transformations that perform edits on a class diagram [Ban87]. Examples are renaming a class, adding new instance variables, and moving a method up the class hierarchy. *Design patterns* are recurring sets of relationships between classes, objects, methods, etc. that define preferred solutions to common object-oriented design problems [Gam95]. The *hot-spot-driven-approach* is based on the identification of aspects of a software program which are likely to change from application to application (i.e. *hot-spots*) [Pre95]. Architectures using abstract classes and template methods are prescribed to keep these hot-

1. We gratefully acknowledge the sponsorship of Microsoft Research, the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226), and the University of Texas at Austin Applied Research Laboratories.

spots flexible.

Refactorings are behavior-preserving program transformations which directly aid in the implementation of new architectures. Primitive refactorings perform simple edits such as adding new classes, creating instance variables, and moving instance variables up the class hierarchy. Compositions of refactorings can create abstract classes, capture aggregation and components [Opd92], extract template and hook methods, and even install design pattern microarchitectures [Tok95]. Although composing refactorings to achieve a desired result may require some planning, this effort is negligible compared to the manual task of identifying all lines of source code affected by a change, performing hand-edits, retesting all fixes, and risking the introduction of new errors.

We are pursuing two approaches to promote refactoring research. The first is to evaluate refactorings of large applications. We believe that we are the first to provide empirical evidence on the usefulness of refactorings when applied to non-trivial applications [Tok99]. In one example, a major architectural change — the splitting of a class hierarchy — is automated[2]. In a second example on a large application (~500K LOCS), approximately fourteen thousand lines of code changes between two code releases are automated with refactorings. (See paper for details.)

A second approach to promoting refactoring research is to demonstrate that common forms of architectural evolution can be automated. This paper catalogs the schema transformations, design pattern restructurings, and hot-spot meta patterns which can be automated with refactorings. By demonstrating broad coverage of common modes of evolution, it is argued that refactorings will be generally useful in the evolutionary maintenance cycle.

A summary of the class diagram notation used throughout the remainder of this paper is presented in Figure 1. Within the main body of text, we use the following conventions:

- **Refactoring** — a refactoring.
- ***AbstractClass*** — an abstract class name.

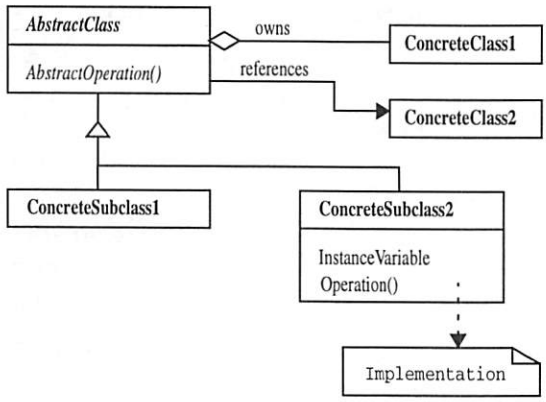


Figure 1: Notation

- **ConcreteClass** — a concrete class name.
- `Method()` — a method or procedure name.
- `Instance_variable` — an instance variable.

2 Refactorings

A *refactoring* is a parameterized behavior-preserving program transformation that automatically updates an application's design and underlying source code. A refactoring is typically a very simple transformation, one that has a straightforward (but not necessarily trivial) impact on application source code. An example is **inherit**[***Base***, **Derived**], which establishes a superclass-subclass relationship between two classes, ***Base*** and **Derived**, that were previously unrelated. From the perspective of an object-oriented class diagram, **inherit** merely adds an inheritance relationship between the ***Base*** and **Derived** classes, but it also checks enabling conditions to determine if the change can be made safely and it alters the application's source code to reflect this change. A refactoring is more precisely defined by (a) a purpose, (b) arguments, (c) a description, (d) enabling conditions, (e) an initial state, and (f) a target state. Such a definition for **inherit**[***Base***, **Derived**] is given in Figure 2. Applying refactorings is superior to hand-coding similar changes because it allow a designer to evolve the architecture of an existing body of code at the level of a class diagram leaving the code-level details to automation.

Banerjee and Kim proposed a set of schema evolutions for evolving object-oriented database schemas [Ban87] and Opdyke proposed a list of primitive refactorings for object-oriented languages [Opd92]. Roberts implements many of these refactorings for the Smalltalk language [Rob97]. In addition to previous refactorings, we have

2. The term *automated* in this paper refers to a refactoring's programmed check for enabling conditions and its execution of all source code changes. The choice of which refactorings to apply is always made by a human.

Name:

Inherit[Base, Derived]

Purpose:

To establish a public superclass-subclass relationship between two existing classes.

Arguments:

Base - superclass name
Derived - subclass name

Description:

Inherit[] makes **Base** a superclass of **Derived**. `vm* ()` represents the unimplemented virtual methods inherited by **Base** subclasses.

Enabling Conditions:

- **Base** must not be a subclass of **Derived** and **Derived** must not have a superclass.
- Subclasses of **Base** must support methods `vm* ()` if objects of that class are created. Otherwise, there will be no implementations for `vm* ()` .
- Initializer lists must not be used to initialize **Derived** objects. Initializer lists must initialize aggregates and aggregates cannot have superclasses [Ell90].
- Program behavior must not depend on the size of **Derived**. Adding a superclass can affect its size.

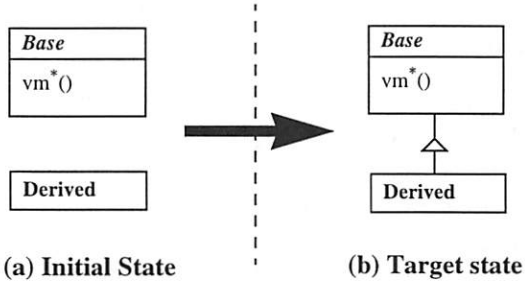


Figure 2: Inherit[Base, Derived] transformation

found that transforming actual applications requires a larger set. We enlarged the set of schema evolutions to include, for example, **substitute**. **Substitute** changes a class' dependency on some class **C** to a dependency on a superclass of **C** [Tok95]. A second new set of refactorings is language-specific. **Procedure_to_method** and **structure_to_class** are used to convert C artifacts to their C++ equivalents. A third set supports the addition of design pattern microarchitectures in evolving programs [Tok95]. An example is **add_factory_method** which creates a method returning a new object and replaces all C++ invocations of "`new Object`" with a call to the method. This refactoring is used to add the Factory Method design pattern [Gam95].

A list of refactorings used in our research is presented in Table 1. Refactorings proposed in previous work are

Schema Refactorings	move_method_across_ object_boundary
add_variable	extract_code_as_method
create_variable_accessor	*declare_abstract_method*
create_method_accessor	*structure_to_pointer*
rename_variable	
remove_variable	C++ Refactorings
push_down_variable	*procedure_to_method*
pull_up_variable	*structure_to_class*
move_variable_across_ object_boundary	
create_class	Design Pattern Refactorings
rename_class	*add_factory_method*
remove_class	*create_iterator*
inherit	*composite*
uninherit	*decorator*
substitute	*procedure_to_command*
rename_method	*singleton*
remove_method	1. [Ban87]
push_down_method	2. [Opd92]
pull_up_method	3. [Tok95]
	4. [Rob97]

Table 1: Object-oriented refactorings

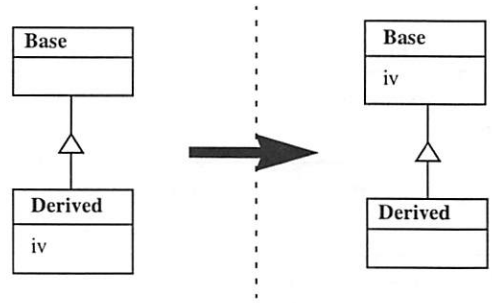


Figure 3.1: Using pull_up_variable to move instance variables "iv" from Derived to Base

noted. Refactorings first implemented by our research for object-oriented software evolution appear in italics.

3 Automatable Modes of Evolution

3.1 Schema Transformations

The database schema for an object-oriented database management system (OODBMS) looks like a class diagram for an object-oriented application. Similarly, OODBMS schema transformations have parallels in object-oriented software evolution. An example schema transformation is moving the domain of an instance variable up the inheritance hierarchy Figure 3.1. This transformation is supported by the refactoring **pull_up_variable** which moves an instance variable to

a superclass.

Banerjee and Kim describe 19 object-oriented database schema transformations of which we implement 12 as automated refactorings[3]:

Description from Banerjee and Kim [Ban87]	Refactoring from Table 1
Adding a new instance variable	**add_variable**
Drop an existing instance variable	**remove_variable**
Change the name of an instance variable	**rename_variable**
Change the domain of an instance variable	**pull_up_variable** and **push_down_variable**
Drop the composite link property of an instance variable[a]	**structure_to_pointer**
Drop an existing method	**remove_method**
Change the name of a method	**rename_method**
Make a class S a superclass of class C	**inherit**
Remove class S as a superclass of class C	**uninherit**
Add a new class	**create_class**
Drop an existing class	**remove_class**
Change the name of a class	**rename_class**

a. A class **A** with an instance variable of class **B** having the *composite link* property specifies that **A** owns **B**. **B** cannot be created independently of **A** and **B** cannot be accessed through a composite link of another object.

Three other useful schema transformations not listed in [Ban87] are:

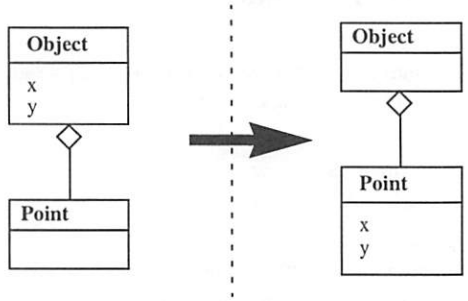


Figure 3.2: Using move_variable_across_object_b move instance variables x and y

Description	Refactoring
Move a variable through a composite link	**move_variable_across_object_boundary (Figure 3.2)**
Move a method through a composite link	**move_method_across_object_boundary**
Change a class' dependency on a class C to a dependency on a superclass S of C	**substitute (Figure 3.3)**

Schema transformations perform many of the simple edits encountered when evolving class diagrams. They can be used alone or in combination to evolve object-oriented architectures.

3. The seven refactorings which are not supported are: changing the value of a class variable, changing the code of a method, changing the default value of an instance variable, changing the inheritance parent of an instance variable, changing the inheritance of a method, adding a method, and changing the order of superclasses. The first three refactorings are not behavior-preserving. The next two are not supported by mainstream object-oriented programming languages. The sixth (adding a method) cannot be automated. The seventh (changing the order of superclasses) is not supported because this research is currently limited to applications without multiple inheritance.

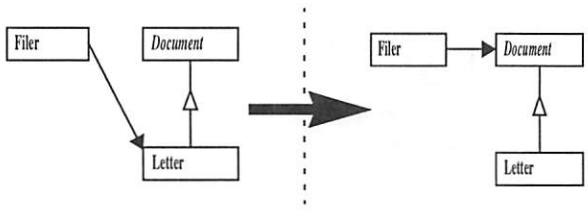


Figure 3.3: Using substitute to change Filer's reference to a Letter to a reference to a Document

3.2 Design Pattern Microarchitectures

Design patterns capture expert solutions to many common object-oriented design problems: creation of compatible components, adapting a class to a different interface, subclassing versus subtyping, isolating third party interfaces, etc. Patterns have been discovered in a wide variety of applications and toolkits including Smalltalk Collections [Gol84], ET++ [Wei88], MacApp [App89], and InterViews [Lin92]. As with database schema transformations, refactorings have been shown to directly implement certain design patterns:

Pattern	Description	Example
Command	Command encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. The **procedure_to_command** refactorings converts a procedure to a command class.	[Tok99]
Factory Method	Factory Method defines an interface for creating an object, but lets subclasses decide which class to instantiate. The **add_factory_method** refactoring adds a factory method to a class.	[Tok95]
Singleton	Singleton ensures a class will have only one instance and provides a global point of access to it. The **singleton** refactoring converts an empty class into a singleton.	[Tok99]

We directly support three additional patterns as refactorings:

Pattern	Description
Composite	Composite composes objects into tree structures to represent part-whole hierarchies. The **composite** refactoring converts a class into a composite class.
Decorator	Decorator attaches additional responsibilities to an object dynamically. The **decorator** refactoring converts a class into a decorator class.
Iterator	Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The **create_iterator** refactoring generates an iterator class.

While design patterns are useful when included in an initial software design, they are often applied in the maintenance phase of the software lifecycle [Gam93]. For example, the original designer may have been unaware of a pattern or additional system requirements may arise that require unanticipated flexibility. Alternatively, patterns may lead to extra levels of indirection and complexity inappropriate for the first software release. A number of patterns can be viewed as automatable program transformations applied to an evolving design. Examples for the following two patterns have been documented:

Pattern	Description	Example
Abstract Factory	Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete class.	[Tok95]

Pattern	Description	Example
Visitor	Visitor lets you define a new operation without changing the classes of the elements on which it operates.	[Rob97]

At least five additional patterns from [Gam95] can be viewed as a program transformations:

Pattern	Description
Adapter	Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Bridge decouples an abstraction from its implementation so that the two can vary independently.
Builder	Builder separates the construction of a complex object from its representation so that the same construction process can create different representations.
Strategy	Strategy lets algorithms vary independently from the clients that use them.
Template Method	Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.

In all cases, we can apply refactorings to simple designs to create the designs used as prototypical examples in [Gam95]. The following sections show how the first two patterns can be automated.

3.2.1 Adapter

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. In the object adapter example from [Gam95] (Figure 3.4), the **TextShape** class adapts **TextView**'s `GetExtent()` method to implement `BoundingBox()`. The adapter can be constructed from the original **TextView** class (Figure 3.5) in five steps:

1. Create the classes **TextShape** and ***Shape*** using **create_class**.
2. Make **TextShape** a subclass of ***Shape*** using **inherit** (Figure 3.6).
3. Add the `text` instance variable to **TextShape** using **add_variable** (Figure 3.7).

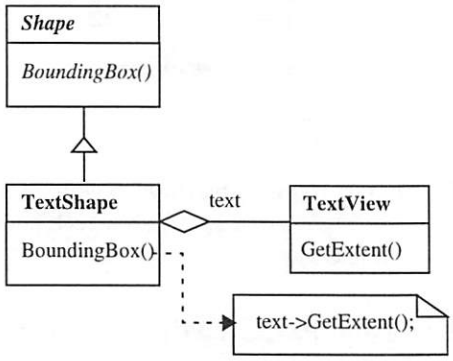


Figure 3.4: TextShape adapts TextView's interface

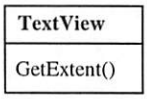


Figure 3.5: Unadapted TextView class

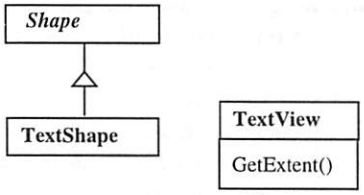


Figure 3.6: Adapter class created

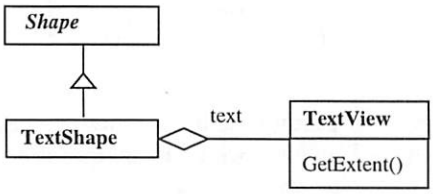


Figure 3.7: Adaptee instance variable added to adapter

4. Create the `BoundingBox()` method which calls `text->GetExtent()` using **create_method_accessor**. **Create_accessor_method** creates a method which replaces calls of the form `instance_variable->method()`.
5. Declare `BoundingBox()` in ***Shape*** using **declare_virtual_method** (Figure 3.4).

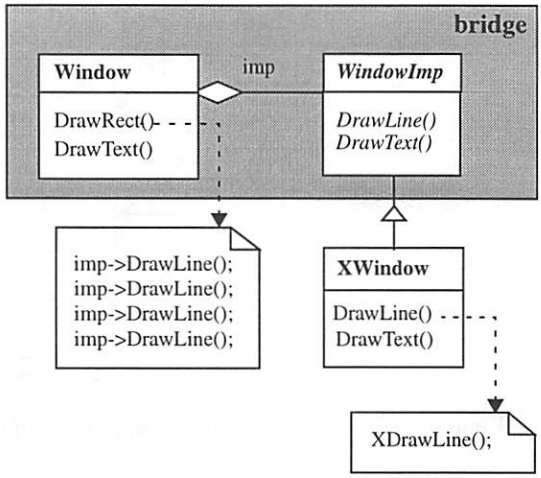


Figure 3.8: Bridge design pattern example

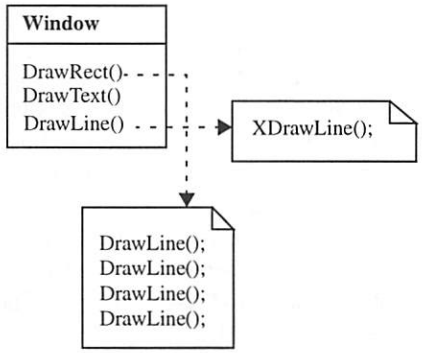


Figure 3.9: Design for a single window system

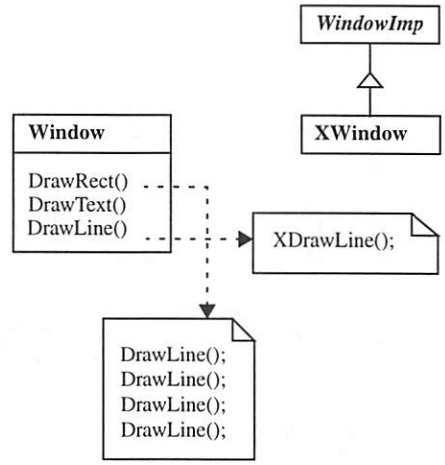


Figure 3.10: Implementor classes created

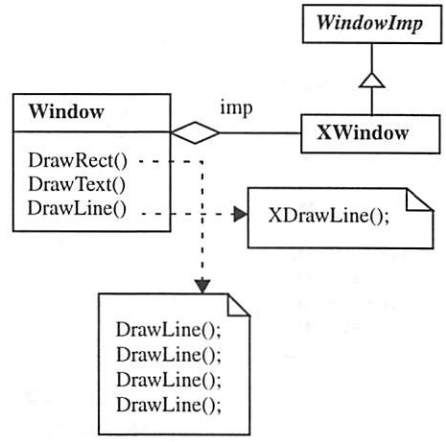


Figure 3.11: Implementor instance variable added to Window

3.2.2 Bridge

Bridge decouples an abstraction from its implementation so that the two can vary independently. In the example from [Gam95] (Figure 3.8), the ***Window*** abstraction and ***WindowImp*** implementation are placed in separate hierarchies. All operations on ***Window*** subclasses are implemented in terms of abstract operations from the ***WindowImp*** interface. Only the ***WindowImp*** hierarchy needs to be extended to support another windowing system. We refer to the relationship between ***Window*** and ***WindowImp*** as a bridge because it bridges the abstraction and its implementation, allowing them to vary independently.

Refactorings can be used to install a bridge design pattern given a simple design committed to a single window system. Figure 3.9 depicts a system designed for X-Windows. This system can be evolved with refactorings to use the bridge design pattern in seven steps:

1. Create classes **XWindow** and ***WindowImp*** using **create_class**.
2. Make ***WindowImp*** a superclass of **XWindow** with **inherit** (Figure 3.10).
3. Add instance variable `imp` to the **Window** class using **add_variable** (Figure 3.11).
4. Move methods `DrawLine()` and `DrawText()` to the **XWindow** class using the refactoring **move_method_across_object_boundary**. These methods are accessed through the `imp` instance variable (Figure 3.12).
5. Declare method `DrawLine()` and `DrawText()` in ***WindowImp*** with **declare_abstract_method**.

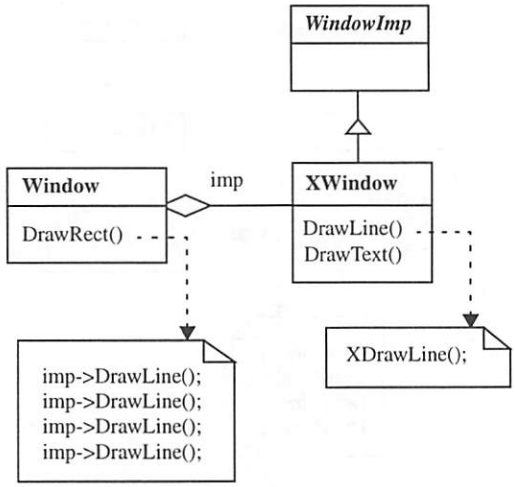


Figure 3.12: Window system specific methods moved to XWindow class

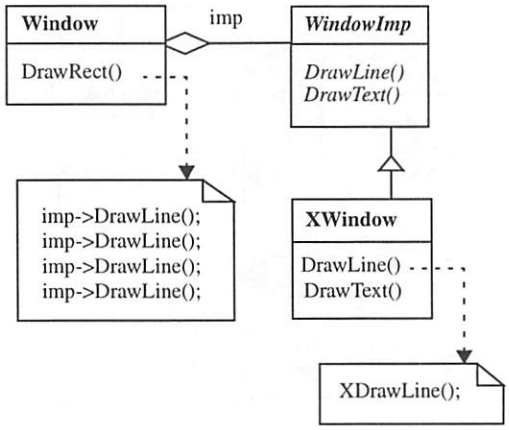


Figure 3.13: Virtual methods declared so that "imp" can be generalized to class *WindowImp*

6. Change the type of instance variable `imp` from **XWindow** to ***WindowImp*** using **substitute** (Figure 0.13).
7. Add a `DrawText()` method to **Window** which calls `DrawText()` in ***WindowImp*** using **create_method_accessor** (Figure 3.8).

The Bridge architecture uses object composition to provide needed flexibility. Object composition is also present in the Builder and Strategy design patterns. The trade-offs between use of inheritance and object composition are discussed in [Gam95, pp. 18-20]. Refactorings allow a designer to safely migrate from statically checkable designs using inheritance to dynamically defined designs using object-composition.

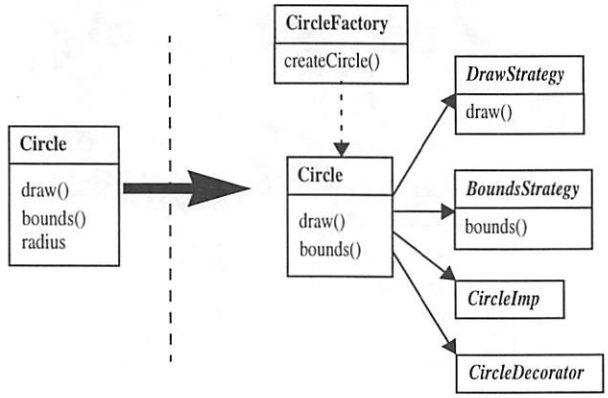


Figure 3.14: Overenthusiastic use of design patterns

3.2.3 Role of Refactorings for Design Patterns

Gamma et. al. note that a common design pattern pitfall is overenthusiasm: "Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible." The example from [Gam96] is displayed in Figure 3.14. Instead of creating a simple **Circle** class, an overenthusiastic designer adds a **Circle** factory with strategies for each method, a bridge to a **Circle** implementation, and a **Circle** decorator. The design is likely to be more complex and inefficient that what is actually required. The migration from a single **Circle** class to the complex microarchitecture in Figure 3.14 can be viewed as a transformation. This transformation is in fact automatable with refactorings[4]. Thus, instead of overdesigning, one can start with a simple **Circle** class and add the Factory Method, Strategy, Bridge, and Decorator design patterns as needed.

Refactorings can restructure existing implementations to make them more flexible, dynamic, and reusable, however, their ability to affect algorithms is limited. Patterns such as Chain of Responsibility and Memento require that algorithms be designed with knowledge about the patterns employed. These patterns are thus considered fundamental to a software architecture because there is no refactoring enabled evolutionary path which leads to their use. Refactorings allow a designer to focus on fundamental patterns when

4. A Circle factory is created [Tok95]. Strategies are added (Section 3.2). The Bridge pattern is applied (Section 3.2.2). Finally, a decorator is added (Section 3.2).

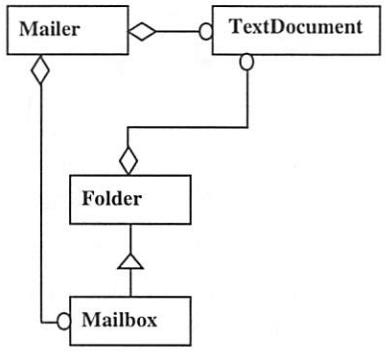


Figure 3.15: Initial state of mailing system

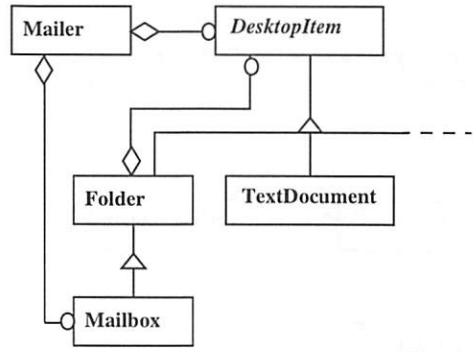


Figure 3.16: Final state of mailing system

creating a new software architecture. Patterns supported through refactorings can be added on an if-needed basis to the current or future architecture at minimal cost.

3.3 Hot-Spot Analysis

The *hot-spot-driven-approach* [Pre94] identifies which aspects of a framework are likely to differ from application to application. These aspects are called *hot-spots*. When a data hot-spot is identified, abstract classes are introduced. When a functional hot-spot is identified, extra methods and classes are introduced.

3.3.1 Data Hot-Spots

When the instance variables between applications are likely to differ, Pree prescribed the creation of abstract classes. Refactorings have repeatedly demonstrated the ability to create abstract classes [Opd93, Tok95, Rob97]. As an example, Pree and Sikora provide a Mailing System case study [Pre95]. Figure 3.15 displays the initial state of its software architecture. In this system, **Folder** cannot be nested, and only **TextDocument** can be mailed. Their suggested architecture is displayed in Figure 3.16. Under the improved architecture, **Folders** can be nested and any subclass of ***DesktopItem*** can be mailed. Refactorings can automate these changes in five steps:

1. Create a ***DesktopItem*** class using **create_class** (Figure 3.17).
2. Make ***DesktopItem*** a superclass of **TextDocument** using **inherit** (Figure 3.18).
3. Generalize the link between **Mailer** and **TextDocument** to a link between **Mailer** and ***DesktopItem*** using **substitute** (Figure 3.19). Subclasses of ***DesktopItem*** can now be mailed.

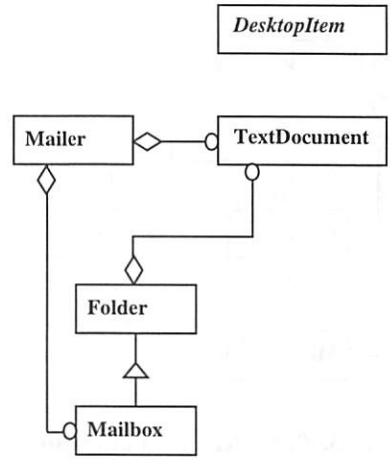


Figure 3.17: Empty TextDocument class created

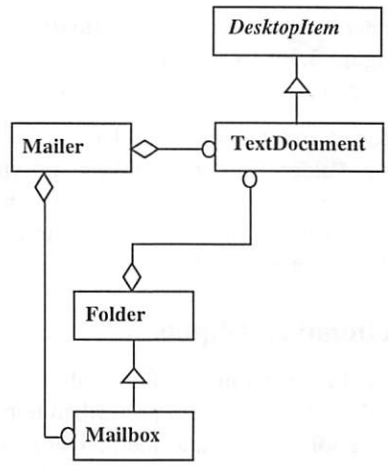


Figure 3.18: TextDocument inherits from *DesktopItem*

4. Generalize the link between **Folder** and

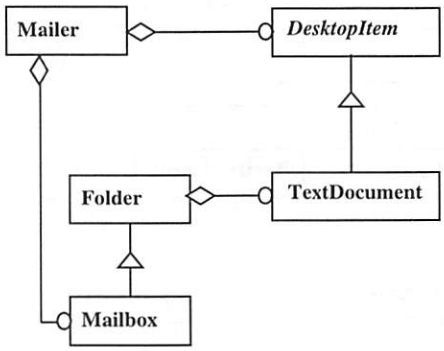


Figure 3.19: Mailer dependency changed from TextDocument to *DesktopItem*

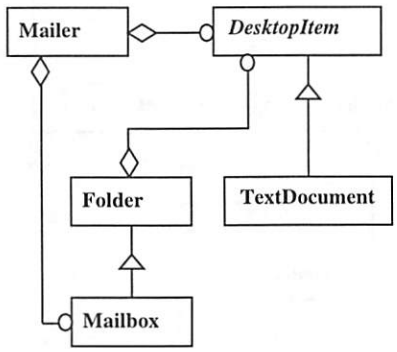


Figure 3.20: Folder can contain any *DesktopItem*

TextDocument to a link between **Folder** and ***DesktopItem*** using **substitute** (Figure 3.20). **Folder** can now contain any ***DesktopItem***.

5. Make **Folder** a subclass of ***DesktopItem*** using **inherit** (Figure 3.16). A **Folder** which can contain a ***DesktopItem*** can now contain another **Folder**.

With the improved architecture, a **Folder** can be nested within another **Folder** and ***DesktopItem*** provides a superclass for adding other types of media to be mailed. These changes which would normally be implemented and tested by hand can be automated with refactorings.

3.3.2 Functional Hot-Spots

For the case of differing functionality, solutions based on *template* and *hook methods* are prescribed to provide the needed behavior. A template method provides the skeleton for a behavior. A hook method is called by the template method and can be tailored to provide different behaviors. Figure 3.21 is an example of a template method and hook method defined in the same class. Different subclasses of ***T*** can override hook

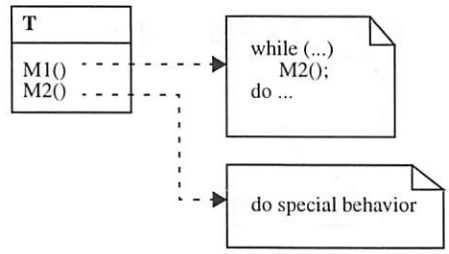


Figure 3.21: Template and hook methods in same class

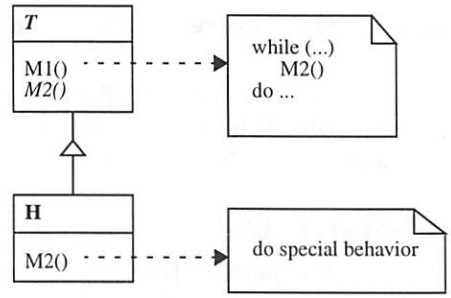


Figure 3.22: Hook method M2() overridden in class H

method M2() which leads to differing functionality in template method M1(). (Figure 3.22). Pree identifies seven meta patterns for template and hook methods: unification, 1:1 connection, 1:N connection, 1:1 recursive connection, 1:N recursive connection, 1:1 recursive unification, and 1:N recursive unification [Pre94]. Refactorings automate the introduction of meta patterns into evolving architectures. The transitions between patterns enabled by refactorings are displayed in Figure 3.23[5]. As examples, we demonstrate support for the first two transitions.

In the unification composition, both the template and hook methods are located in the same class (Figure 3.21). The behavior of the template is changed by overriding the hook method in a subclass (Figure 3.22). An architecture with no template or hook methods can be transformed to use the unification meta pattern (transition 1 from Figure 3.23). Consider the class

5. We consider the 1:N connection composition to be fundamental to an architecture. For this pattern, a template object is linked to a collection of hook objects. This implies that the template method has knowledge about how to use multiple hook methods and thus cannot be derived from the 1:1 connection composition in which the template method is coded for a single hook method.

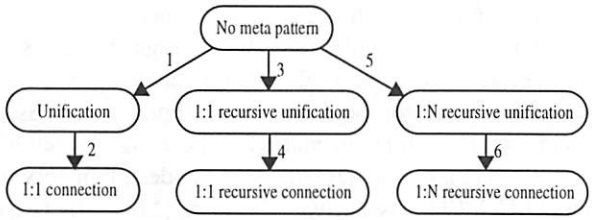


Figure 3.23: Hot-spot meta pattern transitions enabled by refactorings

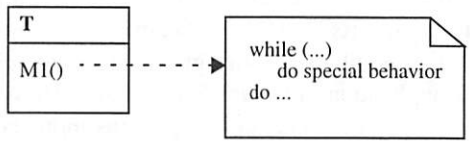


Figure 3.24: Method M1() calls a special behavior which differs for each application

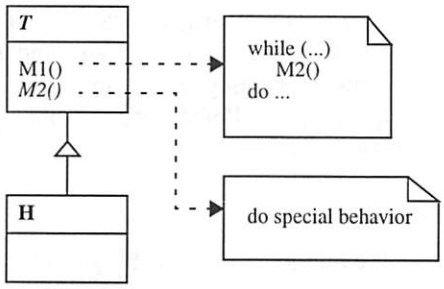


Figure 3.25: Hook class created

diagram in Figure 3.24 with class **T** having method M1() which calls some special behavior. A hook method can be added with refactorings in one step:

1. Create a hook method M2() which executes the special behavior using **extract_code_as_method** (Figure 3.21). **Extract_code_as_method** replaces a block of code with a call to a newly created method which executes the block.

In the new microarchitecture, general behavior is contained in template method `M1()` while special behavior is captured by hook method `M2()`. To extend the architecture, subclasses of **T** override `M2()` to provide alternative behaviors for `M1()`. The extended structure can be added in four steps:

1. Create class **H** using **create_class**.

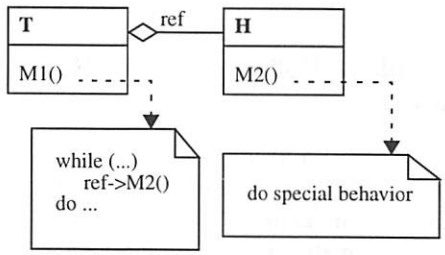


Figure 3.26: 1:1 connection

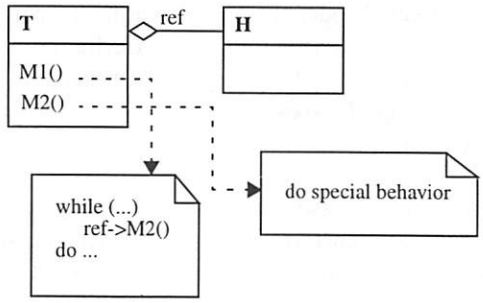


Figure 3.27: Connection to H object created

2. Make *T* a superclass of **H** using **inherit** (Figure 3.25).
3. Make *M2()* overridable by the subclasses of *T* using **declare_abstract_method**.
4. Move the implementation of M2() into **H** using **push_down_method** (Figure 3.22).

As a second example, we support the transition from unification to 1:1 connection (transition 2 from Figure 3.23). Consider the 1:1 connection meta pattern which stores the hook method in an object owned by the template class (Figure 3.26). Behavior can be changed at run-time by assigning a hook object with a different behavior to the template class. 1:1 connection can be automated in three steps using the unification pattern (Figure 3.21) as a starting point.

1. Create class **H** using **create_class**.
2. Add an instance variable `ref` to **T** with **add_variable** (Figure 3.27).
3. Move M2() to class **H** using **move_method_across_object_boundary** (Figure 3.26).

The behavior of template method `M1()` can now be altered dynamically by pointing to different hook class

objects with different implementations of `M2()`. Other transitions in Figure 3.23 are similarly supported.

3.3.3 Role of Refactorings for Hot-Spot Analysis

The hot-spot-driven-approach provide a comprehensive method for evolving designs to manage change in both data and functionality. Pree notes that "the seven composition meta patterns repeatedly occur in frameworks." Thus, we expect an ongoing need to add meta patterns to evolving architectures. The addition of meta patterns is currently a manual process. Conditions are checked to ensure that a pattern can be added safely, lines of affected source code are identified, changes are coded, the system is tested to check for errors, any errors are fixed and the system is retested. Retesting continues until the expected likelihood of an error is sufficiently low.

This section demonstrates that most meta patterns can be viewed as transformations from a simpler design. Refactorings automate the transition between designs granting designers the freedom to create simple frameworks and add patterns as needed when hot-spots are identified.

4 Related Work

Griswold developed behavior-preserving transformations for structured programs written in Scheme [Gri91]. The goal of this system was to assist in the restructuring of functionally decomposed software. Software architectures developed using the classic structured software design methodology [You79] are difficult to restructure because nodes of the structure chart which define the program pass both data and control information. The presence of control information makes it difficult to relocate subtrees of the structure chart. As a result, most transformations are limited to the level of a function or a block of code.

Object-oriented software architectures offer greater possibilities for restructuring. Bergstein defined a small set of object-preserving class transformations which can be applied to class diagrams [Ber91]. Lieberherr implemented these transformations in the Demeter object-oriented software environment [Lie91]. Example transformations are deleting useless subclasses and moving instance variables between a superclass and a subclass. Bergstein's transformations are object preserving so they cannot add, delete, or move methods or instance variables exported by a class.

Banerjee and Kim identified a set of schema transformations which accounted for many changes to evolving object-oriented database schema [Ban87]. Opdyke defined a parallel set of behavior-preserving transformations for object-oriented applications based on the work by Banerjee and Kim, the design principles of Johnson and Foote [Joh88], and the design history of the UIUC Choices software system [May89]. These transformations were termed *refactorings*. Roberts developed the Smalltalk Refactory Browser which implements many of these refactorings [Rob97].

Tokuda and Batory proposed additional refactorings to support design patterns as targets states for software restructuring efforts [Tok95]. Refactorings are shown to support the addition of design patterns to object-oriented applications [Tok95, Rob97, Sch98]. Winsen used refactorings to make design patterns more explicit [Win96]. Tokuda and Batory demonstrated that refactorings can automate significant (greater than 10K lines of code) changes when applied to real applications [Tok99].

A number of tools instantiate a design pattern and insert it into existing source code [Bud96, Kim96, Flo97]. Instantiations are not necessarily refactorings, so testing of any changes may be required. Florijn and Meijers check invariants governing a pattern and repairs violations when possible. Refactorings do not have this pattern-level knowledge.

5 Summary

Architectural evolution is a costly yet unavoidable consequence of a successful application. One method for reducing cost is to automate aspects of the evolutionary cycle when possible. For object-oriented applications in particular, there are regular patterns by which architectures evolve. Three modes of architectural evolution are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. Many evolutionary changes can be viewed as program transformations which are automatable with object-oriented refactorings. Refactorings are superior to hand-coding because they check enabling conditions to ensure that a change can be made safely, identify all lines of source code affected by a change, and perform all edits. Refactorings allow architectural evolution to occur at the level of a class diagram and leave the code-level details to automation.

Architectures should evolve on an if-needed basis:

- "Complex systems that work evolved from simple systems that worked." — Booch
- "Start stupid and evolve." — Beck

Refactorings directly address the need to evolve from simple to complex designs by automating many common design transitions. We believe that the majority of all object-oriented applications undergoes some form of automatable evolution. The broad scope of supported changes indicates that refactorings can have a significant impact when applied to evolving designs. This claim is validated with real applications in [Tok99] where many hand-coded changes between two major releases of two software systems are automated.

The limiting factor barring the widespread acceptance of refactoring technology appears to be the availability of production quality refactorings for the two most popular object-oriented languages: C++ and Java. Our current research identifies implementation issues for C++ [Tok99].

References

[Ban87] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[Ber91] P. Bergstein. Object-Preserving Class Transformations. In *Proceedings of OOPSLA '91*, 1991.

[Bud96] F. J. Budinsky et.al., Automatic Code Generation from Design Patterns. In *IBM Systems Journal*, Volume 35, No. 2, 1996.

[Ell90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[Gam93] E. Gamma et. al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings, ECOOP '93*, pages 406-421, Springer-Verlag, 1993.

[Gam95] E. Gamma et.al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gam96] E. Gamma et. al. *TUTORIAL 29: Design Patterns Applied.* OOPSLA '96 Tutorial, 1996.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[Gri91] W. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis. University of Washington. August 1991.

[Joh88] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, pages 22-35, June/July 1988.

[Kim96] J. Kim and K. Benner. An Experience Using Design Patterns: Lessons Learned and Tool Support, *Theory and Practice of Object Systems*, Volume 2, No. 1, pages 61-74, 1996.

[Lie91] K. Lieberherr, W. Hursch, and C. Xiao. *Object-Extending Class Transformations.* Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[Flo97] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings, ECOOP '97*, pages 472-495, Springer-Verlag, 1997.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Opd93] W. F. Opdyke and R. E. Johnson. Creating Abstract Superclasses by Refactoring. In *ACM 1993 Computer Science Conference*. February 1993.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128-138, March 1979.

[Pre94] W. Pree. Meta Patterns — A Means for Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings, ECOOP '94*, Springer-Verlag, 1994.

[Pre95] W. Pree and H. Sikora. *Application of Design Patterns in Commercial Domains*. OOPSLA '95 Tutorial 11, Austin, Texas, October 1995.

[Pre92] R. Pressman. *Software Engineering A Practitioner's Approach*, McGraw Hill, 1992.

[Rob97] D. Roberts, J. Brant, R. Johnson. A Refactoring Tool for Smalltalk. In *Theory and Practice of Object Systems*, Vol. 3 Number 4, 1997.

[Sch98] B. Schulz et. al. On the Computer Aided Introduction of Design Patterns into Object-Oriented

Systems. In *Proceedings of the 27th TOOLS Conference*, IEEE CS Press, 1998.

[Tok95] L. Tokuda and D. Batory. Automated Software Evolution via Design Pattern Transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[Tok99] L. Tokuda and D. Batory. *Evolving Object-Oriented Designs with Refactorings*. University of Texas, Department of Computer Science, Technical Report TR99-09, March 1999.

[Wei88] A. Weinand, E. Gamma, and R. Marty. ET++ -- An Object-Oriented Application Framework in C++. In Object-Oriented Programming Systems, Languages, and Applications Conference, pages 46-57, San Diego, California, September 1988.

[Win96] Pieter van Winsen. *(Re)engineering with Object-Oriented Design Patterns*. Master's Thesis, Utrecht University, INF-SCR-96-43, November, 1996.

[You79] E. Yourdon and L. Constantine. *Structured Design*. Prentice Hall, 1979.

The Design and Implementation of **Guaraná**

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

Abstract

Several reflective architectures have attempted to improve meta-object reuse by supporting composition of meta-objects, but have done so using limited mechanisms such as Chains of Responsibility. We advocate the adoption of the Composite pattern to define *meta-configurations*. In the meta-object protocol (MOP) of **Guaraná**, a *composer* meta-object can control *reconfiguration* of its component meta-objects and their interactions with base-level objects, resolving conflicts that may arise and establishing meta-level *security policies*.

Guaraná is currently implemented as an extension of *Kaffe OpenVM*™, a free implementation of the Java[1] Virtual Machine. Nevertheless, most design decisions presented in this paper can be transported to other programming languages and MOPs, improving their flexibility, reconfigurability, security and meta-level code reuse. We present performance figures that show that it is possible to introduce run-time reflection support in a language like Java without much impact on execution speed.

1 Introduction

Object-oriented design is based on abstraction and information hiding (encapsulation). These concepts have provided an effective framework for the management of complexity of applications. Within this framework, software developers strive to obtain applications that are highly coherent and loosely coupled. Unfortunately, object orientation alone does not address the development of software that can be easily adapted.

The concept of open architectures [6, 7] has been proposed as a partial solution to the problem of creating software that is not only modular, well-structured, but also easier to adapt. Open architectures encourage a modular design where there is a clear separation of *policy*, that is, *what* a module has been designed for, from the *mechanisms* that implement a policy, that is, *how* a policy is materialized. The implementation of system-oriented mechanisms such as concurrency control, distribution, persistence and fault-tolerance can benefit from this approach to software construction.

Computational reflection [13, 21] (henceforth just reflection) has been proposed as a solution to the problem of creating applications that are able to maintain, use and change representations of their own designs (structural or behavioral). Reflective systems are able to use self-representations to extend and *adapt* their computation. Due to this property, they are being used to implement open software architectures. In reflective architectures, components that deal with the processing of self-representation and management of an application reside in a software layer called *meta-level*. Components that deal with the functionality of the application are assigned to a software layer called *base-level*. In object-oriented reflective systems, meta-level objects that implement management policies are called *meta-objects*.

Due to their inherent structure, the existing reflective architectures and MOPs may induce developers to create complex meta-objects that, in an all-in-one approach, implement many management aspects of an application or, alternatively, to construct coherent but tightly coupled meta-objects. Both alterna-

[1] Java is a trademark of Sun Microsystems, Inc.

tives make reuse, maintenance and adaptation of an application harder, especially of its meta-level, the layer in which most of the adaptations tend to occur in an open architecture.

In contrast, **Guaraná** [20] allows meta-objects to be combined through the use of *composers*. Composers [17] are meta-objects that can be used to define arbitrary policies for delegating control to other meta-objects, including other composers. They provide the glue code to combine meta-objects, and to resolve conflicts between incompatible ones. The use of composers encourages the separation of the *structure* of the meta level from the *implementation* of individual management aspects.

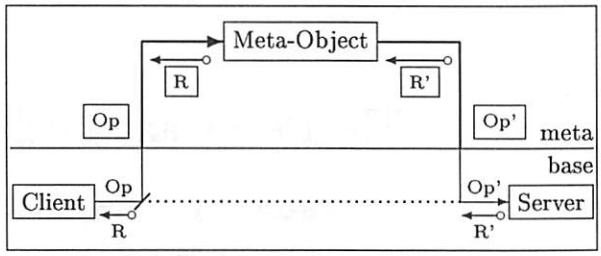
Our implementation of **Guaraná**, based on a Java interpreter that supports just-in-time compilation, has shown that it is possible to introduce interception mechanisms, essential for the deployment of behavioral reflection, with a small overhead. We believe that this overhead is a minor drawback, when compared with the flexibility introduced by our MOP.

This paper is structured as follows. In the next section, we discuss some related works. In Section 3, we present the reflective architecture of **Guaraná**. Section 4 contains a short description of our implementation of this architecture, extending a freely-available Java Virtual Machine. In Section 5, we present some figures about the impact of **Guaraná** on the performance of applications. Section 6 lists some possible future optimizations for our implementation of **Guaraná**. Finally, in Section 7, we summarize the main points of the paper.

2 Related Work

The development of generic mechanisms for the composition of meta-objects is still in its initial stages. OpenC++ [2] does not provide direct support for composition. MOOSTRAP [16] and MetaXa [9] (formerly known as MetaJava) support sequential composition of *similar* meta-objects. We say that meta-objects are similar if they implement the same interface.

Apertos [23] and CodA [14] assign aspects of base-level execution, such as sending, receiving and



When a Client requests an operation Op from a Server object, the operation is intercepted, reified (represented as an object) and presented to a Meta-Object. It may choose to deliver a different operation Op' to the Server, obtaining the result R', that is also reified. Having delivered an operation or not, it must reply with a result R, that is unreified and returned to the Client.

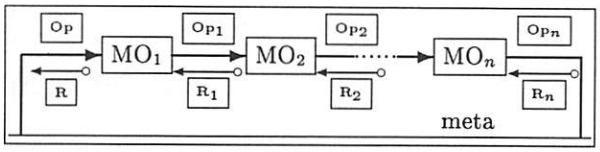
Figure 1: Basic interception.

scheduling operations, to specialized, dissimilar meta-objects. A pre-determined set of aspects can be extended, through intrusive modification of the implementation of the meta-objects responsible for them. We consider this a primitive mechanism of composition, that fails in the general case, because the modifications are very likely to clash.

Several run-time MOPs have been designed so that, when a meta-object is requested to *handle* a reified operation (for example, a method invocation), it is *obliged*, by the design of the MOP, to return a valid result for the operation (typically the value returned by the method), as shown in Figure 1. The meta-level computation that yields the result can include or not the delivery of the operation to the base-level object.

This design implies that the only way to combine the behavior of meta-objects is by arranging for one meta-object, say MO_1, to forward operation handling requests to another, say MO_2, delegating to MO_2 the responsibility for computing the result of the operation. Only after MO_2 returns a result will MO_1 be able to observe and/or to modify it.

Given such a protocol, meta-objects are likely to be organized in a Chain of Responsibility [5, chapter 5], so that each meta-object delegates operation handling requests to its successor, as depicted in Figure 2. The last element of the chain is either the base-level object [9] or a special meta-object that delivers operations to it [16]. We argue that this design presents some serious drawbacks:



Given the basic interception mechanism of Figure 1, meta-objects can only be composed with a Chain of Responsibility [5, chapter 5], a sequential delegation pattern.

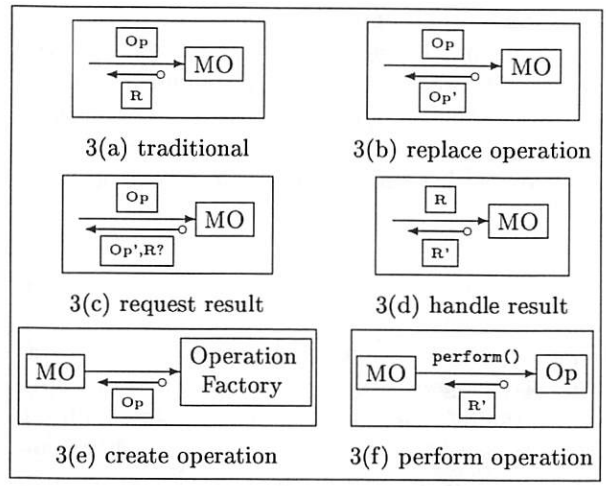
Figure 2: Chain of Meta-Objects.

- it is intrusive upon the meta-object implementation, in the sense that a meta-object must *explicitly* forward operations to its successor;
- it forbids multiple meta-objects from concurrently handling the same operation, because, at a given moment, at most one meta-object can be responsible for producing a result or delivering the operation to the base level;
- it forces meta-objects to receive the results of operations they handled, even if they are not interested in them;
- the order of presentation of results is necessarily the reverse order of the reception of operations, even though different (possibly concurrent) orderings might be more appropriate or efficient, according to the semantics and the requirements of the application;
- it is impossible to mediate interactions between meta-objects and base-level objects with an adaptor capable of resolving conflicts that might arise when multiple meta-objects are put to work together.

Even AspectJ [11, 12], an aspect-oriented programming [8] extension of Java, lacks the possibility of introducing such an adaptor to manage conflicting weaves of aspects so that they can coexist.

3 The Reflective Architecture of Guaraná

The problems presented in the end of Section 2 are solved in the MOP of **Guaraná** by splitting the meta-level processing associated with a base-level operation in the following steps:



*This figure presents the basic MOP of **Guaraná**: although a meta-object is allowed return a result when requested to handle an operation (a), it may prefer to return an operation to be performed (b), with or without an indication that it is interested in its result (c). If it is, it will be presented the result after the execution of the operation (d). Meta-objects can use operation factories to create operations (e) that can replace other operations (b,c) or be performed as stand-alone ones (f).*

Figure 3: Operations and Results.

1. If the target object of the operation is associated with a meta-object, the kernel of **Guaraná**—the entity that implements the MOP— intercepts and reifies the operation and requests the meta-object to handle it; otherwise, no meta-level computation occurs, reducing the overhead for non-reflective objects.
2. A meta-object may produce a result for an operation, as in Figure 3(a). In this case, the meta-level processing terminates by unreifying the result as if it had been produced by the execution of the intercepted operation.
3. However, the meta-object is not required to reply with a result. This permission is essential because *it cannot deliver the operation to the base-level object.* Instead, it should *reply* with an operation to be delivered to the base level (Figure 3(b)) —usually the operation it was requested to handle— and with an indication of whether it is interested in observing and/or modifying the result of the operation (Figure 3(c)).

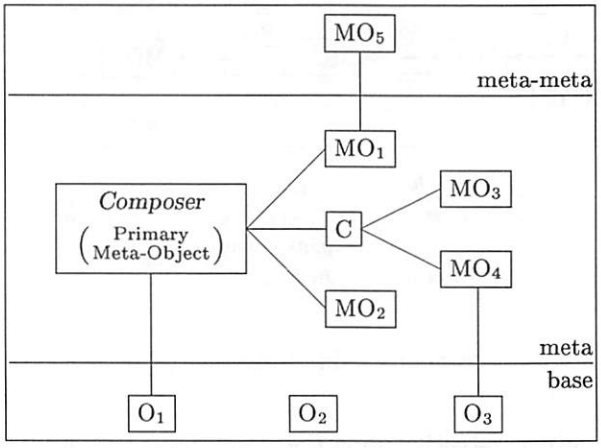
4. Finally, the operation is delivered to the base level, and its result may or may not be presented to the meta-object, depending on its previous reply (Figure 3(d)). If it had requested for permission to modify the result, it may now reply with a different result for the operation.

Replacement operations can be created in the meta-level using *operation factories*, as in Figure 3(e). Operation factories allow meta-objects to obtain privileged access to the base-level objects they manage. Stand-alone operations can also be created with operation factories, and then *performed*, i.e., submitted for interception, meta-level processing and potential delivery for base-level execution, as in Figure 3(f).

We have been able to define *composers* by separating operation handling from result handling, implemented in two distinct methods, namely, handle operation and handle result. A composer is a meta-object that delegates operations and results to multiple meta-objects, then composes their replies in its own replies. For example, a composer can implement the chain of meta-objects presented before, but in a way that one meta-object does not have to keep track of its successor. Another implementation of composer may delegate operations and/or results concurrently to multiple meta-objects, or refrain from delegating an operation to some meta-objects if it is aware they are not interested in that operation.

In **Guaraná**, at any given moment, each object can be directly associated with at most one meta-object, called its *primary meta-object*. If there is no such association, operations addressed to that object are not intercepted, and we say that the object is not reflective at that moment.

The fact that **Guaraná** associates a single (primary) meta-object with an object keeps the design of the interception mechanism simple. Since the primary meta-object can be a composer, as can any meta-object it delegates to, multiple meta-objects can reflect upon an object. These meta-objects form a Composite pattern [5, chapter 4] that we call the *meta-configuration* of that object (Figure 4), a potentially infinite hierarchy of composition that is orthogonal to the well-known infinite tower of meta-objects [13].



The meta-configuration of O_1 is elaborate: a composer, called its primary meta-object, delegates to three other meta-objects, one of which is a composer itself, and delegates to two other meta-objects. O_2 is not associated with any meta-object, so its operations are not intercepted; it is not reflective. O_3 shares MO_4 with O_1. MO_1 is a reflective meta-object, since it has its own (meta-)meta-configuration.

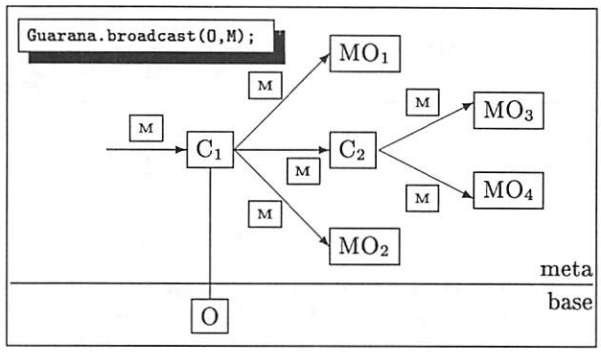
Figure 4: Meta-configurations.

3.1 Meta-configuration management

Guaraná presents two additional features that enforce the separation of concerns between the base level and the meta level: (i) the meta configuration of an object is completely hidden from the base level and even from the meta level itself; and (ii) the initial meta-configuration of an object is determined by the meta-configurations of its creator and of its class, a mechanism we call *meta-configuration propagation*.

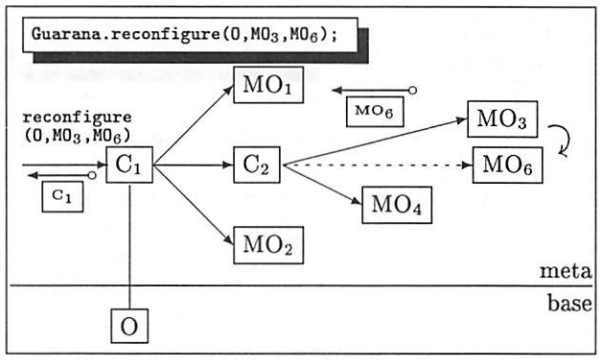
The first design decision implies that there is no way to find out what is the primary meta-object associated with an object. It is possible, however, to send arbitrary *messages* and *reconfiguration requests* to the components of the meta-configuration of an object, through the kernel of **Guaraná**.

Messages can be used to extend the MOP of **Guaraná**, as they allow meta-objects to exchange information even if they do not hold references to each other. Meta-objects that do not understand a message are supposed to ignore it, and composers are expected to forward messages to their components, as in Figure 5. The kernel operation that



Any object M (for message) can be sent to the primary meta-object of an object O. Composers usually forward messages to their components. For non-reflective objects, this request is ignored.

Figure 5: Broadcasting a message.

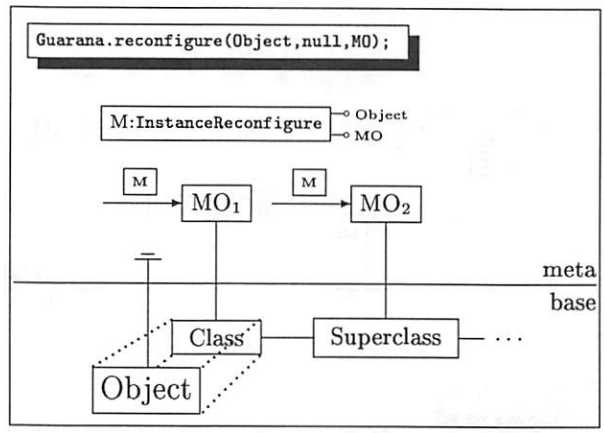


A request to replace MO_3 with MO_6 in the meta-configuration of O was issued. As the request descends the composition hierarchy, it reaches the target meta-object. In this case, it agrees to be replaced, by returning the proposed meta-object. A meta-object must return itself in order to ignore the request, as C_1 does, otherwise the returned meta-object will replace it.

Figure 6: Dynamic reconfiguration.

implements this mechanism is called broadcast.

A reconfiguration request (Figure 6) carries a pair of meta-objects, suggesting that the first meta-object (MO_3) should be replaced with the second (MO_6) in the meta-configuration of object `O`. A special value (`null`) can be used to refer to the primary meta-object. It is up to the existing meta-configuration to decide whether the request is acceptable or not. However, if the base-level object is not reflective, an InstanceReconfigure mes-



The `null` meta-object can be used as an alias for the primary meta-object in reconfiguration requests. When the object is not reflective, the meta-configuration of its class will be given the opportunity to affect the proposed meta-configuration of the instance. An InstanceReconfigure message will carry the proposed meta-object, so that meta-objects of the object's class(es) may modify it. The remaining meta-object will become the object's primary meta-object.

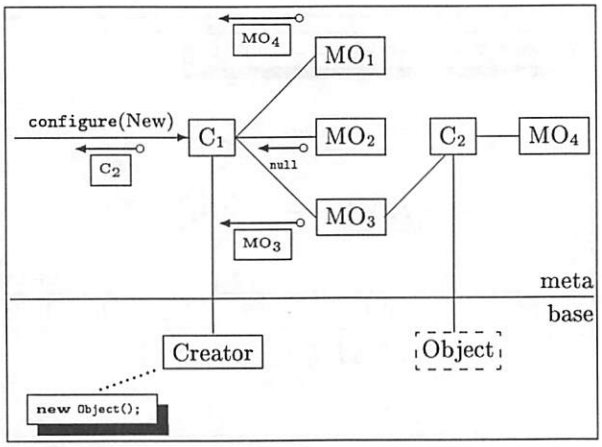
Figure 7: Reconfiguration of a non-reflective object.

sage is broadcast to the meta-configurations of its class and of its superclasses, as depicted in Figure 7. Their components can modify the suggested meta-configuration, for example, forcing it to remain empty.

In most object-oriented programming languages, creating an object consists of two steps: (i) allocating storage for the object, possibly initialized with default values, then (ii) invoking its constructor. We say that these steps are performed by the *creator* of the object.

Meta-configuration propagation takes place between these two steps in **Guaraná**. The primary meta-object of the creator is responsible for providing a meta-object for the new object. It may return `null`, a different meta-object or even itself, as a meta-object can belong to multiple meta-configurations. A composer is expected to forward this request to its components and to create a composer that delegates to the meta-objects returned by them, as in Figure 8.

After meta-configuration propagation, the kernel of **Guaraná** broadcasts a NewObject message to the



When a reflective object instantiates another object, its meta-configuration may propagate to the new object before the object is initialized. In fact, the meta-configuration does not have to propagate as a whole: in the picture, only MO_3 was effectively propagated; MO_2 was discarded, whereas MO_1 named MO_4 to occupy its place in the meta-configuration of the new object. C_1 created a new composer to delegate to MO_4 and MO_3.

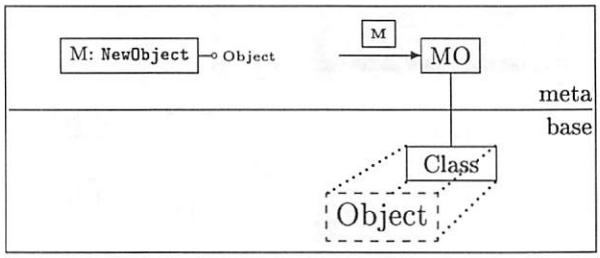
Figure 8: Meta-configuration propagation.

meta-configuration of the class of the new object, so that its meta-objects can try to reconfigure it, as shown in Figure 9. Finally, the object is constructed, but the constructor invocation will be intercepted if the new object has become reflective.

3.2 Support for proxy objects

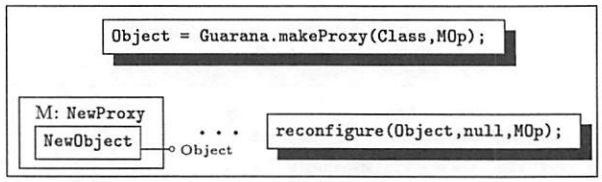
Guaraná provides a mechanism that allows *proxy* objects to be created from the meta level, *without* invoking their constructors. In addition to the traditional use of a proxy, namely, for representing an object from another address space, a proxy can be used to reincarnate an object from persistent storage, to migrate an object, etc.

When a proxy is created, as in Figure 10, the kernel of **Guaraná** broadcasts to the meta-configuration of its class a NewProxy message, a subclass of NewObject. A proxy will usually be given a meta-configuration that prevents operations from reaching it, but it may be transformed in a real object by its meta-configuration, through constructor invocation or direct initialization.



After meta-configuration propagation, the meta-configuration of the class of a new object is notified about the new instance, with a NewObject message, so that it can try to affect the meta-configuration of its instances, by issuing reconfiguration requests.

Figure 9: NewObject messages.



It is possible to request the creation of a proxy object of any class. As soon as the proxy is created, a NewProxy message, subclass of NewObject, is broadcast to the meta-configuration of the class, so that it can take control over the proxy before the proposed meta-object does. Afterwards, a reconfigure request is automatically issued to try to install the proposed meta-object as the primary meta-object of the proxy.

Figure 10: Proxy objects.

3.3 Security

Another advantage of the MOP of **Guaraná** is its concern with security. The hierarchy of composition can be used to limit the ability of a meta-object to affect a base-level object. For example, a composer may decide not to present an operation to a meta-object, or to ignore results or replacement operations it produces. The composer can withhold a message to a component, reject a meta-object produced by a component at a reconfiguration or propagation request, or provide restrictive operation factories to its components, thus limiting their ability to create operations. Furthermore, since the identity of the primary meta-object of an object is not exposed, the hierarchy cannot be subverted.

4 Implementation

We had originally intended to implement **Guaraná** in 100% Pure Java, either by writing an extended Java interpreter in Java or by introducing interception mechanisms through a bytecode preprocessor. The first alternative was discarded because it could imply poor performance and difficulties in handling native methods [22]. A bytecode preprocessor implementation was not possible either, due to restrictions imposed by the Java bytecode verifier [10] and the impossibility to rename native methods, needed in order to ensure their interception.

Therefore, we have decided to implement **Guaraná** by modifying the Kaffe OpenVM™, an open-source Java Virtual Machine. Most of **Guaraná** is coded in Java, but the Java Virtual Machine has suffered a very minor and localized modification, in order to provide for interception of operations. The performance impact due to the modification was quite small (Section 5) especially when compared to the benefit of transparent interception of method invocations, field and array accesses, object instantiation, and monitor primitives.

The Java Programming Language, however, has not been modified. Thus, any Java program, compiled with any Java compiler, will run on our implementation, within the limitations of the Kaffe OpenVM, the most portable existing Java Virtual Machine. We consider this aspect of **Guaraná** yet another benefit of our approach as programmers will be able to use the reflective mechanisms provided to adapt Java programs originally implemented in the absence of any concern with reflection, even without access to the program's source code. This is possible by starting a meta-application to set up meta-configurations of application classes and objects before the application runs. Then, the meta-application starts the application, but it can still control it through interception, meta-configuration propagation and instance reconfiguration messages. **Guaraná** also provides probe meta-objects that can be helpful for figuring out the behavior of certain objects, so that they can be properly configured.

The MOP of **Guaraná** can also be implemented in other object-oriented programming languages, or even upon existing reflective platforms, as an extension to their built-in MOPs. However, some particular features of **Guaraná** may be difficult to duplicate, if some design decisions for the target language or MOP conflict with those of **Guaraná**.

Java 1.1 was an excellent choice as a target language for **Guaraná**, because it already provides some reflective properties, such as the ability to represent classes, methods and fields as objects (i.e., these elements of the language are reified), so that it is possible to navigate a class hierarchy (introspection) and even interact with objects using the Java Core Reflection API to reflectively invoke methods and to get or set the value of fields. However, such interactions are restricted by the language access control rules, mimicked at run-time. In Java 2, access control can be supressed for particular instances of Methods and Fields, allowing an instance of class that is able to perform the access to supply privileged access to other objects. Other than that, the Reflection API allows an object to perform only the operations that it would have been allowed to perform directly in source code, i.e., access control is based on class permissions.

Guaraná builds upon these features, introducing mechanisms for interception, that are missing in Java, and per-object (as opposed to per-class) security mechanisms, so that meta-objects can obtain privileged access to objects they control.

5 Performance

We have run some performance tests to try to evaluate the impact of introducing reflective capabilities into a Java interpreter. Like the other few papers in the literature on reflection that provide performance data, we have preferred to evaluate the overhead of reflection on each particular operation, instead of running standard benchmarks. In fact, there are no standard benchmarks to evaluate the impact of reflection. Existing general-purpose benchmarks usually focus on optimization of complex patterns of control flow, which would not be affected by the introduction of interception for objects operations, and calculations on large arrays, which would incur a huge overhead.

Our tests have been performed on four different platforms, listed in Table 1. On the Solaris platforms, the tests were run in real-time scheduling mode, so as to ensure that no other processes would affect the measured times. On the GNU/Linux plat-

Table 1: Description of the platforms.

This table describes the platforms on which the performance tests were run.

Tag	Description
i586	100 MHz Pentium running RedHat Linux 5.1
i686	233 MHz Pentium Pro running RedHat Linux 5.0
spu1	167 MHz SPARC Ultra 1 running Solaris 2.6
spu2	200 MHz SPARC Ultra Enterprise 2 running Solaris 2.5

forms, this scheduling mechanism was not available, so we just ensured that the tested hosts were as lightly loaded as possible.

On each host, we have run the same Java program, compiled with Sun JDK's Java compiler, without optimization, to prevent method inlining. The produced bytecodes were executed by different interpreters under different configurations.

We have used **Guaraná** 1.4.1 and the snapshot of Kaffe 1.0.b1 distributed with it, using the JIT compiler and the interpreter engines. Kaffe and **Guaraná** were compiled with EGCS 1.1b, with default optimization levels. The program used to perform the tests was the one distributed with **Guaraná** 1.4.1.

For each configuration, we have timed several different operations, described in Table 2. Each operation was timed by running it repeatedly inside a loop, after running it once outside the loop, before starting the timer. This ensures that, before the loop starts, any JIT compilation has already taken place, all the data and code was brought into the cache and, unless the test involves object allocation, the garbage collector will not run.

This inner loop is run repeatedly, with the iteration count being adjusted at every outer iteration, aiming at a running time longer than 1 second. Since the operations that read the clock at the beginning and at the end of each inner loop take less than 1 microsecond to run, and the clock resolution is 1 millisecond, a total running time of 1 second is enough to elliminate any effects they might have in the outcome of the tests.

Table 2: Description of the tests.

This table describes the operation(s) performed within a loop in our performance tests.

Operation	Description
`emptyloop`	No reflective operation.
`synchronized`	Empty block `synchronized` on an arbitrary object.
`invokestatic`	Invoke an empty `static` method that takes no arguments and returns void.
`invokespecial`	Invoke a non-`static` `private` do-nothing method that returns void and takes only the implicit `this` as argument. The same bytecode is used to invoke constructors and, in some cases, `final` methods.
`invokevirtual`	Invoke an empty method that takes only the implicit `this` as argument, and returns void. Dynamic binding, performed with a dispatch table, occurs before interception test.
`invokeinterface`	Invoke the same method, but through an object reference of `interface` type. Dynamic binding is much slower in this case.
`getstatic`	Load a `static int` field into a variable.
`putstatic`	Store a zero-valued variable in a `static int` field.
`getfield`	Load a non-`static int` field into a variable.
`putfield`	Store a zero-valued variable in a non-`static int` field.
`arraylength`	Load the length of an array of `int` into a variable.
`iaload`	Load the first element of an array of `int` into a variable.
`iastore`	Store a zero-initialized variable in the first element of an array of `int`.
`println`	Print the line ``Hello world!'' to System.err, which was redirected to `/dev/null` before starting the Virtual Machine. It is a first attempt to estimate the overall impact of introducing interception abilities.
`compile`	Compile the test program itself. Section 5.1 contains a detailed description and analysis.

Table 3: Overhead on interpreter.

No interception occurs in these tests, they just measure the overhead imposed on the interpreter to introduce the ability *to intercept operations.*

Operation	i586	i686	spu1	spu2
emptyloop	−41%	−15%	−0%	−0%
synchronized	−0%	+1%	+0%	+4%
invokestatic	+13%	+0%	+4%	−8%
invokespecial	+30%	+8%	+38%	−10%
invokevirtual	+17%	−0%	+7%	−9%
invokeinterface	−3%	−7%	+20%	−10%
getstatic	−3%	−2%	+20%	−0%
putstatic	−23%	−3%	+24%	+4%
getfield	−22%	−2%	+19%	−0%
putfield	−26%	−2%	+25%	+6%
arraylength	−18%	−9%	+2%	+12%
iaload	−64%	−6%	+1%	−0%
iastore	−14%	−3%	+1%	+1%
println	+6%	+4%	+3%	−2%
compile	+5%	+2%	−2%	−3%

Table 4: Overhead on JIT compiler.

No interception occurs in these tests, they just measure the overhead imposed on the JIT compiler and the code it produces to introduce the ability *to intercept operations.*

Operation	i586	i686	spu1	spu2
emptyloop	+0%	+1%	+0%	+0%
synchronized	+12%	+10%	+27%	+3%
invokestatic	+91%	+20%	+23%	+34%
invokespecial	+119%	+8%	+19%	+28%
invokevirtual	+30%	+158%	−6%	+0%
invokeinterface	+7%	+2%	+3%	+2%
getstatic	+68%	+148%	+163%	+163%
putstatic	+180%	+97%	+90%	+90%
getfield	+293%	+86%	+149%	+149%
putfield	+103%	+96%	+66%	+66%
arraylength	+258%	+86%	+140%	+150%
iaload	+191%	+98%	+55%	+95%
iastore	+236%	+55%	+41%	+45%
println	+45%	+6%	+5%	+12%
compile	+36%	+42%	+32%	+29%
compile-JIT	+105%	+112%	+81%	+54%
compile-diff	+16%	+17%	+20%	+20%

The inner-loop iteration count starts at 1, and is repeatedly multiplied by 10 until it is large enough to be measurable with the clock resolution. As soon as this happens, the elapsed time and the iteration count start to be used to estimate the running-time of an iteration. If the total elapsed time of an execution of the inner loop is longer than one second, the estimate is the final result of the test. Otherwise, it is used to compute the iteration count for the next execution of the inner loop, aiming at a total execution time of 1100 milliseconds.

With the exception of the tests `println` and `compile`, this mechanism selected an iteration count between 50,000 and 100,000,000, for the final execution of the inner loop of each test. In the case of `println`, the iteration count was never smaller than 500. The `compile` test was run stand-alone, not within this framework.

Each test case was run 50 times on each configuration and platform, and the average times of the runs were used to compute the relative overheads presented in Table 3 and Table 4. Although we have introduced the ability to intercept operations, no actual interception took place during those tests.

Table 5: Total `compile` time.

These are the total execution times of the `compile` test for each configuration. They were used to calculate the lines `compile` in Table 3 and Table 4.

(times are in seconds)

Configuration	i586	i686	spu1	spu2
Kaffe JIT	17	5.1	9.1	7.5
Guaraná JIT	23	7.2	12	9.6
Kaffe interpreter	30	9.2	13	11
Guaraná interpreter	32	9.4	13	10

5.1 The `compile` test

As an additional effort to measure the performance impact of the introduction of interception ability, we have measured the execution time for the Java compiler to translate the test program to Java bytecodes. The averaged execution times are presented in Table 5.

On short-running applications like this, most of the time is spent on virtual machine initialization and JIT compilation, not on running the applica-

Table 6: JIT compilation time for `compile` test.

These are the times spent on JIT compilation during the execution of the `compile` test. They were used to compute the values in the `compile-JIT` line of Table 4.

(times are in seconds)

Configuration	i586	i686	spu1	spu2
Kaffe JIT	3.9	1.3	1.8	1.9
Guaraná JIT	8.0	2.8	3.3	2.9

Table 7: Net `compile` time

These are the differences between total execution time (`compile`) and JIT compilation time (`compile-JIT`), i.e., the times spent on execution of the JIT compiled code. They were used to compute the values in the `compile-diff` line of Table 4.

(times are in seconds)

Configuration	i586	i686	spu1	spu2
Kaffe JIT	13	3.8	7.3	5.5
Guaraná JIT	16	4.5	8.8	6.7

tion itself. The virtual machine start-up, for example, involves executing very large array initialization methods, whose JIT-compilation wastes a lot of memory and CPU cycles, because these methods are executed only once.

Although a complex program, involving several similar classes, is being compiled, Table 6 shows that more than 50% of the total time was spent on JIT-compiling Java Core classes and the Java compiler itself. Therefore, the actual overhead in execution time, at least for long-running applications, is much smaller.

Table 7 presents the differences between the total time and the JIT-compilation time, that represents the time spent on running the actual application, i.e., the compiler. Long running applications, that repeatedly execute the same methods, should present a reflection overhead similar to the relative overhead of this table.

Table 8: Interception time, interpreter.

*This table presents the interception time of various operations in the **Guaraná** interpreter, with a do-nothing meta-object. Field operations refers to `static` and non-`static` field reads and writes. Array operations involve array length reads and array elements reads and writes.*

(times are in milli*seconds)*

Configuration	i586	i686	spu1	spu2
synchronized	0.92	0.30	0.42	0.35
invokestatic	0.59	0.17	0.22	0.18
invokespecial	0.65	0.17	0.23	0.19
invokevirtual	0.65	0.17	0.24	0.19
invokeinterface	0.67	0.18	0.24	0.19
field operations	0.60	0.16	0.21	0.17
array operations	0.56	0.15	0.21	0.17

Table 9: Interception time, JIT compiler.

*This table presents the interception time of various operations in the **Guaraná** JIT compiler, with a do-nothing meta-object. Other operations refers to all field and array operations.*

(times are in milli*seconds)*

Configuration	i586	i686	spu1	spu2
synchronized	0.55	0.018	0.33	0.25
invokestatic	0.30	0.099	0.20	0.15
invokespecial	0.32	0.10	0.18	0.14
invokevirtual	0.33	0.11	0.20	0.15
invokeinterface	0.33	0.11	0.19	0.15
other operations	0.3	0.09	0.17	0.13

5.2 Intercepting operations

We have also performed some tests involving actual interception, using a do-nothing meta-object to intercept the operation that is the subject of each test. The absolute time spent on the interception of a single operation is presented in Table 8, for the interpreter, and in Table 9, for the JIT compiler.

It is worth noting that each `synchronized` block involves two operations, one that enters the monitor of an object and another that leaves it. Since both are intercepted, the interception time is increased. Additional details are available elsewhere [18].

5.3 Overall discussion

In certain combinations of platform and engine, an operation executes faster on **Guaraná** than on the corresponding combination without it. This is quite hard to explain, since **Guaraná** always executes at least as much code as Kaffe does. The tests have been verified so as to ensure that the results are correct, and the generation of the tables from the test results is mostly automated, so there is little place for human error. The better performance can be attributed to factors such as improved fast-RAM cache hit ratio or alignment issues.

The overhead introduced by interception on the interpreter engine is mostly small, because the interpreter is usually orders of magnitude slower than the test for existence of a meta-object. The JIT, however, is severely affected by increased register pressure and additional register spilling and reloading. JIT-compilation costs have increased too, as our tests have shown, but they have only affected the figures of the `compile` test. In all other cases, we ensure that a method is JIT-compiled before we start timing its execution.

Although the interception code has introduced moderate penalties for invoking `static` and `private` methods, the most common kind of invocation (non-`final`) causes a very small overhead, except on `i686`, and `interface` invocations are almost not affected at all.

The bad results for some invocation bytecodes on one `x86` platform but not on the other is unexpected, considering that it executes *exactly* the same machine code on both. It looks like these tests introduce pathological pipeline stalls or branch prediction errors that degrade performance, since the average penalty, measured in `compile-diff`, is very similar on both `x86` platforms, and much lower than most of the individual penalties.

On the other hand, the bad results for all load and store operations on the JIT engines are expected, since these instructions can usually be executed in one or two machine-level instructions, and in **Guaraná** they require at least one more register and two instructions to test for the presence of a meta-object. Fortunately, in object-oriented applications, field and array operations are usually intertwined with method invocations and object creations. Since the latter operations incur a much smaller penalty, and they are one order of magnitude slower than the former ones, the net performance penalty may be acceptable, as the introduction of reflective capabilities may pay off.

It is worth noting that, although we have introduced the ability to intercept object creation, we have not been able to measure the effect of this addition, due to the impredictability of the garbage collector. Anyway, the overhead is known to be negligible, since a single test was introduced in a rather complex function coded in C.

6 Future optimizations

The reflection overhead on the interpreter is quite small. Furthermore, the interpreter is much slower than the JIT compiler, so there is not much point in trying to optimize it any further. For the JIT code, there is little hope for similarly small overheads, though.

One approach we had considered would be to implement all operations, even field and array ones, as invocations of dynamically generated JIT-compiled code. Then, instead of having to test the meta-object reference before performing an operation, an extended dispatch table would contain pointers to these JIT-generated functions, on non-reflective objects, or to interceptor functions, in the case of reflective objects.

However, we do not think this solution would do very well: first, because we would have to look up the dispatch table before executing every single operation, as in a virtual method invocation, and the absolute time for a virtual method invocation is much larger than non-virtual method invocation, so we would end up increasing the cost of most operations, instead of reducing it.

Furthermore, invoking a function requires saving most registers on some ABIs, but this is not required when contents of memory addresses are loaded directly, as field and array operations are currently implemented. In fact, because of Kaffe's inability to carry register allocation information across basic blocks, the fact that **Guaraná** introduces basic blocks in field or array operations forces registers to be stored in stack slots because it *might* be necessary to invoke an interceptor function. A promising

optimization involves improving the register allocation mechanism so as to propagate register allocation information along the most frequently used control flow, that is the one without interception, and move the burden of spilling and reloading registers into the not-so-common case in which interception must take place. This would decrease the cost of both branches, because they currently save all registers and mark them all as unused before they join to proceed to the next instruction. Furthermore, if the JIT compiler ever gets smarter with regard to global register allocation, the additional branches introduced by **Guaraná** will not get it confused.

There is another optimization, that is much harder to implement within Kaffe, but that could reduce the overhead of loops and methods that make heavy access of a particular object or array. The test for the existence of a meta-object could be performed before entering the loop or starting the sequence, and different versions of the code would be generated: one, in which no meta-object test is performed for that object, and another in which the test is performed in every iteration, because the meta-object may change. This optimization is based on a similar proposal for optimizing array reference checking [15]. Unfortunately, this kind of optimization can only be performed if no method invocation nor interception could possibly occur within the loop or sequence, so as to ensure that reconfiguration does not take place within the same thread. Even in this case, other threads might reconfigure the object or array while the code runs, so synchronization operations must also be ruled out, because, by definition of the Java Virtual Machine Specification [10], they flush a local cache a thread might maintain. But it may still be worth the effort for array and field operations, given that the overhead imposed on them is still large.

7 Conclusions

Our research on computational reflection was initially motivated by our willingness to verify the use of MOPs as a tool for structuring and building environments for fault-tolerant distributed programming. We intended to design and implement a library like **MOLDS** [19], a library of *reusable* and *combinable* meta-level components useful for distributed applications, such as persistence, distribution, replication and atomicity.

Unfortunately, none of the existing reflective architectures supported composition of meta-objects in a way that fulfilled our needs. Therefore, we started the development of **Guaraná**. This paper is an effort to convey the positive and negative aspects of this experience.

Guaraná provides a powerful and secure mechanism to combine meta-objects into dynamically modifiable, elaborate meta-configurations. In addition to enforcing a clear separation between the reflective levels of an application, the MOP of **Guaraná** improves reuse of meta-level code by defining a meta-object interface that eases flexible composition. Furthermore, it suggests a separation of concernts between meta-objects, that implement meta-level behavior, from composers, that define policies of composition and organization.

The implementation of the reflective architecture of **Guaraná** required some modifications in a Java interpreter, but not in the Java programming language. Thus, any program created and compiled with any Java compiler will run on our implementation, and it will be possible to use reflective mechanisms in order to extend them.

Our modifications have reduced the speed of the interpreter, but we believe the flexibility introduced by the reflective capabilities outweighs this inconvenience. Furthermore, the performance impact analysis has revealed the current hot spots in the interception mechanisms. We expect to reduce this impact by implementing the suggested optimizations.

Now that we have **Guaraná**, we are concentrating our efforts on the design and implementation of **MOLDS**. The interaction of the various mechanisms foreseen for **MOLDS** will fully demonstrate the power of our MOP. Meanwhile, other projects based on **Guaraná** are demonstrating its flexibility and ease of use. Tropyc [1] is a pattern language for the domain of cryptography, that is currently using **Guaraná** in order to transparently introduce cryptographic mechanisms in electronic commerce applications. The composition strategy of **Guaraná** has also supported the implementation of the Reflective State Pattern and of its adaptation to the domain of fault tolerance [3, 4].

A last evidence of the usefulness of our approach is the possibility of creating a reflective ORB by simply running a 100% Pure Java ORB in **Guaraná**. By doing this, we provide to the users of the ORB

the ability to create reflective middleware and applications, with a development cost close to zero.

The experience with the design and implementation of **Guaraná** and related applications allows us to conclude that initiatives by the software industry to build software that is highly adaptable and reusable should incorporate MOPs as flexible as, and at least as efficient as the one we have described.

A Obtaining Guaraná

Additional information about **Guaraná** can be obtained in the Home Page of **Guaraná**, at the URL `http://www.dcc.unicamp.br/~oliva/guarana/`. The source code of its implementation atop of the *Kaffe OpenVM*, on-line documentation and full papers are available for download. **Guaraná** is *Free Software*, released under the GNU General Public License, but its specifications are open, so non-free clean-room implementations are possible.

B Acknowledgments

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grants 95/2091-3 for Alexandre Oliva and 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*). Additional support is provided by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), for the PRONEX programme and for a PhD scholarship for Alexandre Oliva.

Islene Calciolari Garcia has helped us very much in reviewing and improving this paper. She also made important contributions to the architecture of **Guaraná**.

Douglas C. Schmidt, from Washington University, St. Louis, has provided us with very useful suggestions for the final version of this paper.

Special thanks to Tim Wilkinson, for having started the development of Kaffe OpenVM and having released it as free software.

References

[1] Alexandre Melo Braga, Cecília Mary Fischer Rubira, and Ricardo Dahab. A system of patterns to cryptographic object-oriented software. In *Pattern Languages of Programs Conference - PLOP'98*, July 1998. TR#WUCS-9825.

[2] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA'95*, volume 30, pages 285–299, October 1995.

[3] Luciane Lamour Ferreira and Cecília Mary Fischer Rubira. Reflective design patterns to implement fault tolerance. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 81–85, Vancouver, BC, Canada, October 1998.

[4] A Reflective Object-Oriented Framework for Developing Dependable Software based on Patterns and Metapatterns. Delano medeiros beder and cecília mary fischer rubira and ricardo dahab. In *28th International Symposium on Fault-Tolerant Computing (FastAbstract)*, June 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch (Designer). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, October 1994.

[6] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[7] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97, LNCS 1241*, Finland, June 1997. Springer-Verlag.

[9] Jürgen Kleinöder and Michael Golm. MetaJava: An efficient run-time meta architecture for Java. In *International Workshop on Object Orientation in Operating Systems - IWOOOS'96*, Seattle, Washington, October 1996. IEEE.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Java Series. Addison–Wesley, January 1997.

[11] Cristina Videira Lopes and Gregor Kiczales. *Aspect-Oriented Programming with AspectJ.* Xerox PARC. http://www.parc.xerox.com/aop/aspectj/tutorial.

[12] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ? In *ECOOP'98 Workshop Reader, LNCS 1543.* Springer-Verlag, 1998.

[13] Pattie Maes. Concepts and experiments in computation reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[14] Jeff McAffer. Meta-level programming with CodA. In *ECOOP'95*, pages 190–214, August 1995.

[15] Samuel P. Midkiff, José E. Moreira, and Marc Snir. Optimizing array reference checking in java programs. Technical Report 21184 (94652), IBM, T.J. Watson Research Division, Yorktown Heights, New York, June 1998.

[16] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA'95*, volume 30 of *ACM SIGPLAN Notices*, pages 316–330, Austin, TX, October 1995.

[17] Alexandre Oliva and Luiz Eduardo Buzato. Composition of meta-objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 86–90, Vancouver, BC, Canada, October 1998.

[18] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, Instituto de Computação, Universidade Estadual de Campinas, September 1998.

[19] Alexandre Oliva and Luiz Eduardo Buzato. An overview of MOLDS: A Meta-Object Library for Distributed Systems. In *Segundo Workshop em Sistemas Distribuídos*, Curitiba, PR, Brazil, June 1998.

[20] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflexive architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, April 1998.

[21] Brian C. Smith. Prologue to "Reflection and Semantics in a Procedural Language". PhD Thesis Prologue, 1985.

[22] Antari Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report SMLI TR-98-64, Sun Microsystems Laboratories, March 1998.

[23] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 414–434, October 1992.

Tuning Branch Predictors to Support Virtual Method Invocation in Java

N. Vijaykrishnan
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, 16802
vijay@cse.psu.edu, http://www.cse.psu.edu/~vijay

N. Ranganathan
Department of Electrical and Computer Engineering
University of Texas at El Paso
El Paso, TX
ranganat@ece.utep.edu, http://www.ece.utep.edu/faculty/webrangan

Abstract

Java's object oriented nature along with its distributed nature make it a good choice for network computing. The use of virtual methods associated with Java's object oriented behavior requires accurate target prediction for indirect branches. This is critical to the performance of Java applications executed on deeply pipelined, wide issue processors. In this paper, we investigate the use of a path history based predictor to accurately determine the target of these virtual methods. The effect of varying the various parameters of the predictor on the misprediction rates is studied using various Java benchmarks. Results from this study show that the execution of Java code will benefit from more sophisticated branch-predictors.

1 Introduction

Java is a class-based object oriented language that is used extensively for building networked applications. Some of the features of Java such as the use of virtual methods, dynamic loading and symbolic resolution that make it suitable for developing networked software applications also slow the execution speed of Java code. In this paper, we focus on addressing the performance issues involved with the use of virtual methods in Java.

The use of virtual methods as the default method invocation mechanism results in the execution of frequent indirect branch executions. The *invokevirtual* JVM bytecode [1] that is used to perform virtual method calls constitutes 5% of the Java bytecodes executed on an average for the benchmarks shown in Table 2. Many of the current JVM implementations such as Sun's JDK interpreter and CACAO Just-in-Time Compiler[2] use a dispatch table to implement the *invokevirtual* bytecode. When a virtual method is invoked, the target address is obtained from a fixed index into the the dispatch table of the current object. Finally, an indirect branch instruction is executed to jump to the fetched target address. Thus, an indirect branch is executed for every virtual method invoked. The accurate prediction of these indirect branches is critical to the performance of Java virtual machine (JVM) implementations executing on deeply pipelined systems. Speculative execution is used in such architectures to avoid the performance loss associated with the execution of branch instructions. Accurate branch predictors are essential to avoid discarding the results of the speculative execution following a misprediction.

Current processors employ a branch target buffer (BTB) based mechanism to predict the indirect branches [12]. The mispre-

diction rates for virtual method calls using the branch target buffer is found to range up to 27% as shown in Table 1 for the studied benchmarks. Previous researchers have successfully used path history information to improve the prediction of direct branches [8, 16]. In this paper, the path history of virtual method calls is used to predict target addresses of virtual method invocations. The path history provides the capability to distinguish between different dynamic executions of the same virtual method. In the path history based predictor, a hashing function of the path history of target addresses and the virtual method call site address is used to index a target cache. The cached entry provides the predicted target address of the virtual call.

The paper is organized as follows. Section 2 discusses the background and motivation for this work. In section 3, the path history based target address predictor is introduced. Next, the experimental strategy and benchmarks used in this study are explained in section 4. The effect of the various parameters of the path-history based target address predictor on the performance of the predictor is studied using the benchmarks in Section 5. Starting from a fully associative target buffer of unlimited size, the parameters are optimized sequentially to account for the hardware constraints such as buffer size and limited associativity. The number of history buffers, the path history length, the number of target address bits, the hashing function, and the buffer structure were the parameters varied. Concluding remarks are provided in Section 6.

2 Background

The problem of target prediction for indirect branches has been investigated for C and C++ programs. Calder and Grunwald proposed a 2-bit strategy for updating the branch target buffer (BTB) [18]. The target address entry in the BTB is updated only when two consecutive predictions at that target address are incorrect. This strategy as opposed to the default strategy of updating the entry on each misprediction was shown to improve the performance. Emer and Gloy present several single-level predictors based on a combination of the values of program counter, stack pointer, register number and stack address [19]. They performed their study on SPECint95 programs.

Previous research has shown the use of correlation information from path history to predict the execution of direct branches [8, 16]. Recently, the path history information has been used to predict indirect branches [17, 20]. Chang, Hao and Patt proposed a target cache that uses the branch history to distinguish different dynamic occurrences of each indirect branch [17]. Their study was performed on select SPECint95 programs. Their work also shows the correlation between higher misprediction rates and slower execution speed. In this paper, we make use of this observation and focus on improving misprediction rates. The object oriented programs in C++ and Java use indirect branches with a much higher frequency than in SPECint95 programs. Target address prediction for indirect branches using a suite of C++ programs and SPECint95 was performed in [20]. Their study investigated the impact of various hardware constraints on the performance of a path history based predictor. Our work uses a similar approach in investigating the indirect branch behavior of Java programs. Unlike the previous efforts, the focus of this work is confined to the target prediction of indirect branches that occur due to the virtual method invocations. The best way to improve the performance of virtual method invocations is to eliminate the virtual calls by inlining or statically binding them [14]. However, only a portion of the calls can be safely bound statically [15]. We identify Java code characteristics that enable the use of path based predictors in identifying the target of the virtual calls.

Since virtual method invocation has been identified as one of the major bottlenecks for the performance of Java code [11, 13], the impact of the various parameters of the path history predictor on prediction accuracy is investigated in this work. It was observed in [11] that the proportion of virtual methods is likely to increase due to the trend towards

Table 1: Misprediction rates using normal and 2-bit replacement strategies

Benchmark	BTB misses (%)	2-bit BTB misses (%)
Javac	4.8	3.9
Javadoc	3.5	2.4
Richards	23.4	27.1
Deltablue	1.7	1.2
Heap	2.6	2.1

A 32K direct mapped BTB was used.

fine-grained object design in Java applications. In such an environment, big objects become many smaller objects. Consequently big methods become many smaller methods. This causes many more method invocations and method invocation increasingly becomes a performance bottleneck. In [13], profiling of various Java benchmarks was done to identify virtual methods as one of the bottlenecks in Java execution. This work also investigated the receiver type locality at virtual method call sites.

Java performance studies have been performed in [9], [10] to investigate the need for architectural support for Java execution. In [10], it was concluded based on their study of Java interpreters that it may be premature to provide hardware support for Java execution. The results of this paper indicate that the micro-architectural resources such as branch predictors can be enhanced to support Java execution. However, it must be noted that the support proposed here is based on the Java language characteristics rather than just the interpreter characteristics analyzed in [10].

3 Path History Based Predictor

In this section, the use of path history to improve the target prediction accuracy is investigated. Path history consists of target addresses of recently executed branches. The history of target addresses provides useful correlation information that can be used to improve the branch prediction accuracy. Virtual method calls in Java programs exhibit correlation among the receiver types at call sites. This is due to the presence of correlation among consecutive call sites as shown in Figure 1. In this example, the shape object *s* invokes a series of virtual calls and the different call sites in the function drag_drop have the same receiver type. Another reason for the correlation is due to a sequence of virtual calls triggered by a single virtual method call and the presence of looping constructs as shown in Figure 2. Here, the invocation of a virtual method to print a string triggers a sequence of method invocations. The use of path history in exploiting such correlation among call sites to predict the destination of virtual calls is investigated in this paper.

```
// This is a drag drop code in GUI based programs
class mouse{
 public void drag_drop(shape &s) {
  s.invalidate_object_area();
  screen.invalidate();
  s.move(new_location);
  s.update_object_area();
  s.repaint();
  screen.update();
 } }
```

Fig 1: Correlation in receiver types among call sites

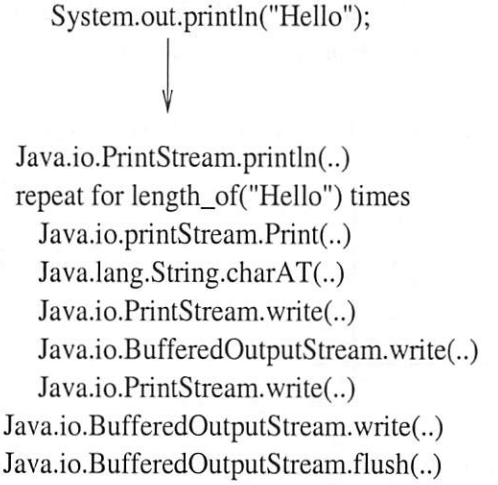


Fig 2: A single virtual method invoking a series of methods

Figure 3 shows the predictor based on the use of path history information. The program counter stores the address of the virtual method call site. An indexing function of the program counter is used to access the path

history information corresponding to the call site. Then a hashing function of the path history information from the history buffers and the program counter is used to form the hashing address. This address is used to index the target buffer to obtain the target address. The various parameters involved in the design of such a predictor include the number of history buffers (n), path history length (p), the number of bits of each target address registered in the history buffer (b), the hashing function, and the structure of the target buffer. The target buffer could either be tagless or a tagged buffer. In a tagless buffer, the hashing address is used to index into the buffer and no tag comparisons are involved. Hence, the mapping of two different hashing addresses in to the same target buffer location are not distinguished. In contrast, the tagged buffer has a tag associated with each entry. These tags help to distinguish different history patterns that map to the same location. The influence of these parameters on the performance of the predictor is studied in the subsequent sections.

Benchmark	Description	T1
Javac	Java compiler	215K
Javadoc	Documentation tool	274K
Richards	O.S task dispatcher [4]	1517K
Deltablue	Constraint solver [4]	12082K
Heap	Garbage collector [3]	151K

T1 - Number of virtual calls executed

Table 2: Description of Benchmarks

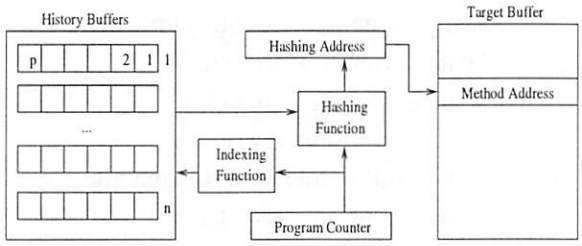


Fig 3: Path history based two level indirect branch prediction

4 Experimental Setup

The following experimental strategy was used in this study. The traces of indirect branches corresponding to the virtual method call sites were obtained through modifications to the JDK 1.0.2 source code. The benchmarks shown in Table 2 were executed using the modified JDK 1.0.2 on a Sparc-20 processor under Solaris 2.5 operating system. The *javac* and *javadoc* benchmarks are large applications with 25,400 and 28,471 lines of code respectively. The *richards* and *deltablue* benchmarks are medium size benchmarks with 410 and 984 lines of code [4]. These two benchmark were chosen as they have been used in earlier studies of polymorphic behavior of object oriented languages [5]. The *heap* benchmark is an 4495 line applet that implements incremental garbage collection.

5 Predictor Parameter Variations

In this section, the effect of varying the parameters involved in the design of the branch predictor is studied. The parameters were optimized sequentially and the following subsections report them in that order. The studied parameters include the number of history buffers (n), path history length (p), the number of bits of each target address registered in the history buffer (b), the hashing function, and the structure of the target buffer.

5.1 Number of History Buffers

The number of history buffers determines the number of virtual method call sites that share their history. When $n = 1$, all the virtual call sites share the same history buffer and the resulting predictor is called as a global history predictor. In contrast, a per-address history predictor keeps a separate history for all virtual method call sites. This is achieved when $n = 2^w$, where w is the word size. When the number of history buffers is between 1 and 2^w, a set of addresses use the same history buffer. The effect of the number of history buffers on misprediction rate was studied by using the most significant bits of the program counter to access the history

buffers. To mask the effects of other parameters of the predictor, a fully associative target buffer of unlimited size was used. Further, all the bits of the target address were registered in the path history buffers and the hashing address was formed by concatenating the selected path history buffer with the program counter.

Figures 4 and 5 show the results of this investigation for *javac* and *richards* benchmarks respectively. The h most significant bits of the program counter select the history buffer corresponding to the call site. The global history predictor is simulated when $h = 0$. In contrast, $h = w$ corresponds to the per-address history predictor. It is observed from the figures that the global path history predictor performs better than those that use per-address or per-set history predictors. For example, the misprediction rates increase from 2.6% for global path history to 4.2% for the per-address scheme using *javac* with a path length of 2. This indicates that the correlation across call sites is more useful than the self history at a call site in predicting the targets. This can be ascribed to the execution of a series of virtual calls corresponding to the invocation of a single virtual call as shown earlier. A global path history can capture the effect of such constructs better than a per-address scheme. Thus, a global path history is used in refining the other parameters of the predictor.

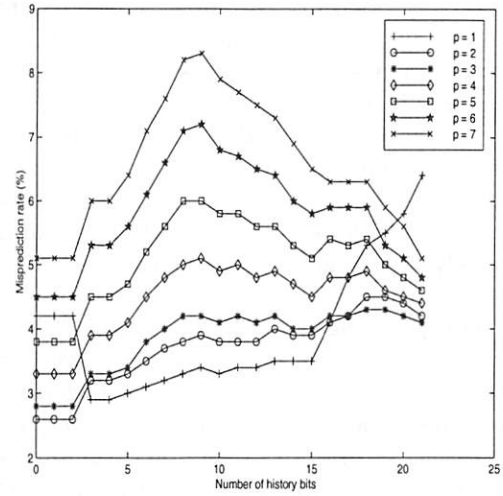


Fig 4: Variation in misprediction rate with number of history buffer sets for *javac*. A fully associative target buffer of unlimited size was used.

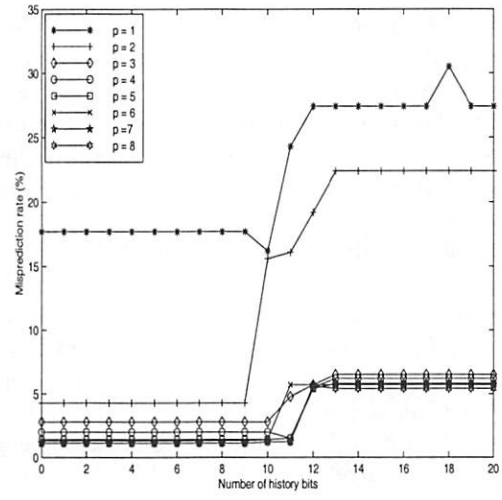


Fig 5: Variation in misprediction rate with number of history buffer sets for *richards*. A fully associative target buffer of unlimited size was used.

5.2 History Path Length

The number of target addresses of the virtual methods registered in each history buffer is called as the history path length, p. When $p = 0$, the two-level path history predictor becomes a single level predictor similar to the BTB strategy. The variation in path length can help in determining whether the correlation among the target addresses of virtual methods is long-term or short-term. The effect of path length variation was studied with a fully associative target buffer of unlimited size along with a global path history. Figure 6 shows the results of this study for the different benchmarks.

The path length affects the misprediction rate in two ways. Firstly, misses occur when the path length is too small to capture a long-term dependence. Secondly, longer paths take a longer time to adapt to branch behavior changes and this results in start-up misses. Thus, a longer path would capture more long term dependence but would have more start-up misses. In contrast, a shorter path fails to capture the long-term regularities in method invocation targets but adapts quickly to changes in branch behavior.

The *javac* benchmark reflects this trade-off clearly. The misprediction rate reduces from 4.6% when the path length is zero to a misprediction rate of 2.6% when the path length is two. In this phase, the effect of capturing more regularities dominates the effect of start-up misses. However, the misprediction rates increase when the path length is increased beyond two, specifically from 2.6% to 5.1% when path length is increased from two to seven. The start-up misses begin to dominate any improvement obtained by capturing virtual method history dependence longer than two. This indicates that most path history patterns used in *javac* have a relatively short period. The *javadoc* benchmark also exhibits a similar behavior.

The *heap* benchmark does not benefit from the path history information. It is observed that the branch target buffer scheme performs better than the predictor with the path history. This is due to the relatively constant target addresses at the call sites in the *heap* benchmark. Hence, the path history information only adds to the start-up misses and does not benefit from capturing any additional regularities. In contrast, the *richards* and *deltablue* benchmarks benefit from long path lengths. The misprediction rates keep decreasing as path lengths increase from 0 to 8. This shows that these benchmarks have a long-term correlation that enables the overshadowing of start-up misses associated with longer paths. These results indicate that the optimum values for the path length differ based on the benchmark.

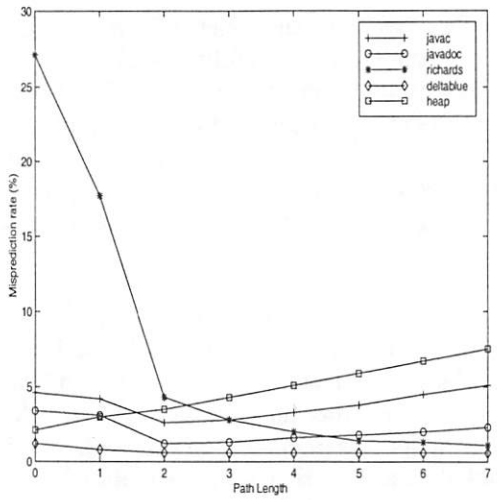


Fig 6: Variation in misprediction rate with path length using global history. A fully associative target buffer of unlimited size was used.

5.3 Path History Compression

The global history pattern along with the branch address stored in the program counter is used to index the target buffer. When all the bits of the history buffer and the program counter are used, the resulting bit pattern is long and is equal to $(p+1)*w$. The number of different path history patterns captured by this hashing address length is $2^{((p+1)*w)}$. However, most programs do not have that many patterns. Thus, the effect of varying the number of bits stored per target address stored in the history buffer on the misprediction rates was investigated. Table 3 shows the results of this investigation for the benchmarks. The least significant bits of the target addresses were used in the history patterns. It is observed that the least significant bits capture more information than the more significant bits. For *javac*, *javadoc* and *deltablue* the misprediction rates decrease when b is increased from 2 to 8 and does not change when bit size of b is increased further. This study shows that registering only the least significant bits of the target address in the history buffer could reduce the bit width of the hashing address without much loss in performance.

Table 3: Effect of history bit compression of misprediction rates

b	Misses (%)			
	Javac	Javadoc	Richards	Deltablue
2	4.7	3.4	23.4	1.6
4	3.7	2.4	25.6	1.3
6	3.3	2.0	6.0	0.8
8	2.6	1.2	6.0	0.6
10	2.6	1.2	6.0	0.6
12	2.6	1.2	6.0	0.6
32	2.6	1.2	4.3	0.6

b is the number of bits from target address used in the path history information.
A path length of two and a fully associative target buffer of unlimited size was used.

5.4 Hashing Function

The effect of the hashing function on the misprediction rates was investigated using limited size tagless target buffers. The hashing function needs to utilize both the path history and the program counter (call site) information effectively. The simplest hashing function is the concatenation scheme shown in Figure 7. Here, h bits of path history information and the program counter are concatenated to form the least significant and most significant bits of the hashing address respectively. Then, the s least significant bits of the hashing address are used to index the target buffer of size 2^s. The contents of the indexed entry provides the predicted target address.

The bit width of the path history buffer, h that constitutes the least significant bits used to index the target buffer was varied and its effect on the misprediction rate was investigated. This was performed to study the relative importance of the path history and program counter information. Figure 8 shows the results of this for an 8K entry target buffer using *javac* for different values of b. It is observed that the misprediction rate decreases, when h increases from 0 to 6. When h is increased further, the misprediction rates increase. Since the 8K target buffer is indexed using a fixed size 13-bit index, the number of bits from the program counter used in the index reduces as the value of h increases. This indicates that the call site location is relatively more important than just the path history information. The target address being primarily determined by the call site location and path history providing only additional information in the prediction accounts for this behavior. Table 4 shows the relative importance of the path history and call site location. It is observed that using only the 13-bit call site location to index the 8K target buffer, a misprediction rate of 4.9% is achieved. The misprediction rate increases to 12.9% when 12 bits of path history and 1-bit of the call site location are used for *javac*.

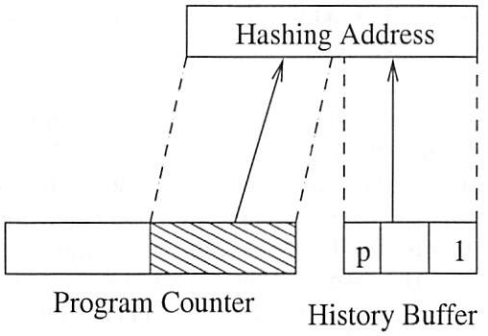


Fig 7: Tagless concatenation scheme with global path history

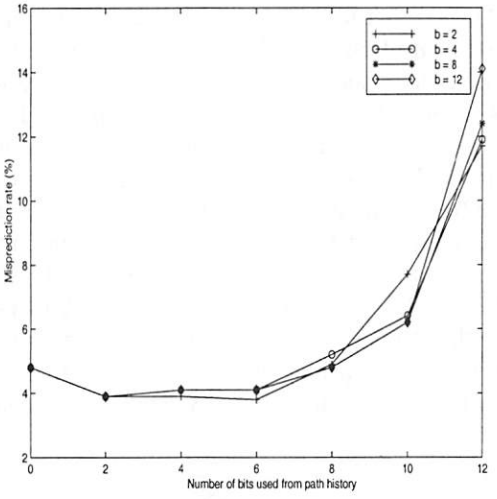


Fig 8: Variation in misprediction rate with concatenated length using tagless concatenation scheme with global path history. An 8K target buffer was used. b refers to number bits per address recorded in the history buffer

In order to utilize both the program counter and the path history bits more effectively for a fixed size target buffer, a XOR hashing scheme shown in Figure 9 was in-

Table 4: Effectiveness of hashing schemes

S	Javac Misses (%)			Richards Misses (%)		
	8K	16K	32K	8K	16K	32K
0	4.9	4.7	4.7	23.4	23.4	23.4
4	4.2	3.8	3.7	4.0	4.0	4.0
6	4.1	3.7	3.5	4.0	4.0	3.9
8	5.2	4.4	3.7	3.8	3.8	3.8
10	6.4	5.0	4.1	8.7	8.0	1.8
12	12.9	7.7	5.6	14.2	8.7	8.7
xor	3.6	3.3	3.2	2.4	2.3	2.0

An entry y in column S refers to the concatenated index formed with y bits of path history and remaining bits from program counter. All schemes register 4 bits of target address in history buffer

vestigated. The XOR hashing function helps in combining more information from the program counter and the path history bits as compared to the concatenation scheme. Here, a bitwise XOR of the program counter and the path history buffer bits is performed to obtain the hashing address. Then, the least significant bits of the hashing address are used to index the target buffer. Two replacement strategies were studied to update the target buffers using the global XOR scheme. These schemes are the same as those studied to update the BTB. It was observed that the 2-bit scheme performs better for the XOR scheme for most of the benchmarks. Figure 10 shows the results for the *javac* benchmark. The 2-bit strategy is referred to as the XOR scheme in the rest of the paper. The effectiveness of the XOR scheme as compared to the concatenation scheme is shown in Table 4. It is observed that for an 8K target buffer, the minimum misprediction rate using the concatenation scheme is 4.1% compared to the 3.6% using the XOR scheme for *javac*.

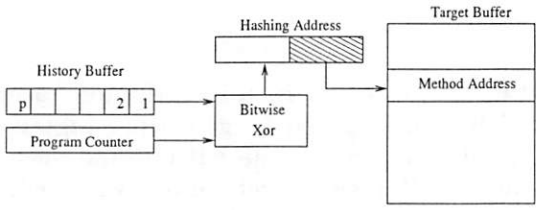


Fig 9: Tagless XOR scheme with global path history

Next, the effect of path length on the misprediction rate of the XOR scheme was investigated. In order to vary the path length,

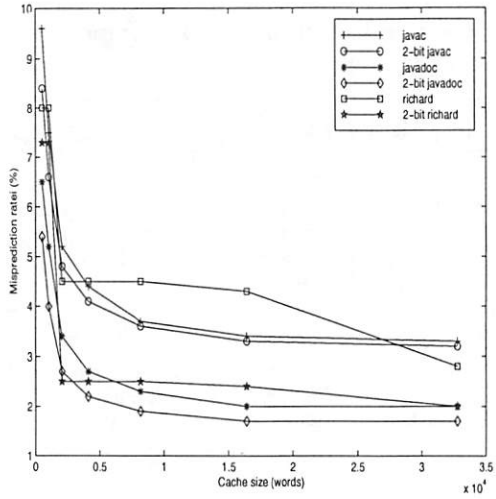


Fig 10: Comparison of normal and 2-bit update schemes using global tagless XOR. All configurations use 4-bit of method address in history buffer

the number of bits b written from each target address into the history buffer was varied. If s bits are required to index the tagless target buffer and p is the path length, b was chosen such that $b * p \leq s$. Figures 11 and 12 show the results of this study for *javac* and *richards* benchmarks respectively. The misprediction rate for *javac* exhibits a similar trend as the fully associative unconstrained target buffer size. It achieves the minimum misprediction rates for a path length of two. For the *richards* benchmark the misprediction rates are the least for a path length of two when target buffer sizes are small(0.5K to 2K). When target buffer size is increased (4K to 32K), the minimum misprediction value is achieved for a path length of three. Thus, a longer path length improves misprediction rate with an increase in the target buffer size. This is due to the greater number of bits of each target address constituting the index portion of the target buffer for a given path length.

5.5 Tagged versus Tagless Target Buffers

We also studied the impact of interference due to the presence and absence of tags with the target buffers. The XOR hashing scheme

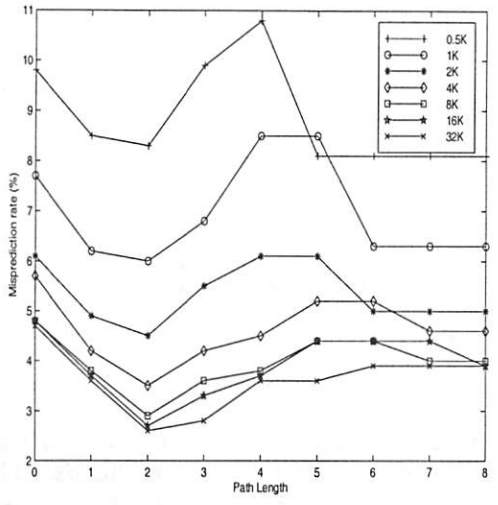


Fig 11: Variation in misprediction rate with path length for *javac* for different target buffer sizes. Uses tagless XOR scheme with global path history

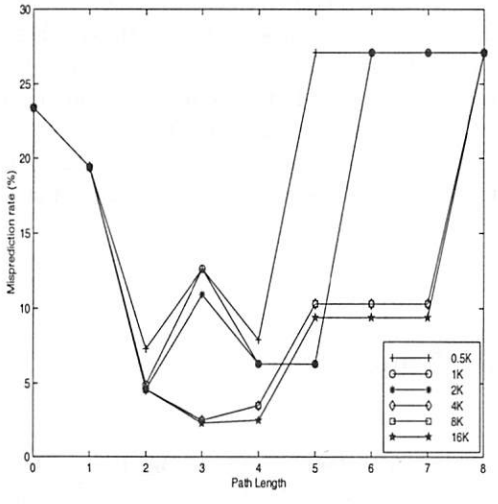


Fig 12: Variation in misprediction rate with path length for *richards* for different target buffer sizes. Uses tagless XOR scheme with global path history

was utilized in studying both the approaches. In the tagless scheme, the target address of the indirect branch is selected using the hashing address to index the target buffer. Since, no tags are associated with each target buffer entry more than one hashing address can map to the same location. Due to this interference, the target of the indirect branch is selected based on the outcome of some other branch path pattern. A positive interference occurs when the when two different patterns that map to the same target location have the same target address. Similarly, when the interference results in more misses it is called as negative interference. A tagged target buffer can be used to eliminate the effects of negative interference.

The impact of the tagged and tagless target buffers was studied for various buffer sizes by varying the associativity and the path length. Figures 13 and 14 shows the variation in misprediction rate using target buffer sizes of 2K and 4K for the tagged and tagless buffers respectively for *javac* and *richards* benchmarks. Additional entries were provided for the tagless case to account for the area overhead in maintaining tags. In these plots, an increase in the number of target address bits in history buffer corresponds to a decrease in path length. It is observed that the misprediction rates decrease with increase in associativity for the tagged buffers. Also, it is observed that there is not a significant improvement when associativity is increased beyond 8. For *javac*, it is observed that the tagless target buffer performs better than the the tagged buffers when $a = 1$ and $a = 2$ for all path lengths. It must be noted that the tagged buffers are useful only when they are able to register the alternate target address when a conflicting path history is identified. Thus, direct mapped tagged buffers perform inherently worse than the tagless buffers as they also do not benefit from positive interference. Hence, higher associativities are required in the tagged caches to benefit from the absence of negative interference.

When path lengths become large (number of target bits in history buffer becomes small), the number of different patterns generated corresponding to an indirect branch increases. Hence, a tagless buffer can benefit from the

positive interference between these different patterns. Thus, it is observed that the tagless buffer performs better than the 4 and 8-way associative tagged buffers for longer path lengths for *javac*. For the *richards* benchmark the effect of positive interference is lesser, since it benefits from longer distinguishing patterns as was observed in Figure 6. Thus, the tagless target buffer performs better than only a direct mapped tagged buffer for all path lengths for the *richards* benchmark. It is also be observed that the the tagged buffers of higher associativity provide a greater improvement in prediction rates for smaller path lengths in both the benchmarks. This indicates that the conflict misses due to short term variations in targets is being reduced by the tagging mechanism.

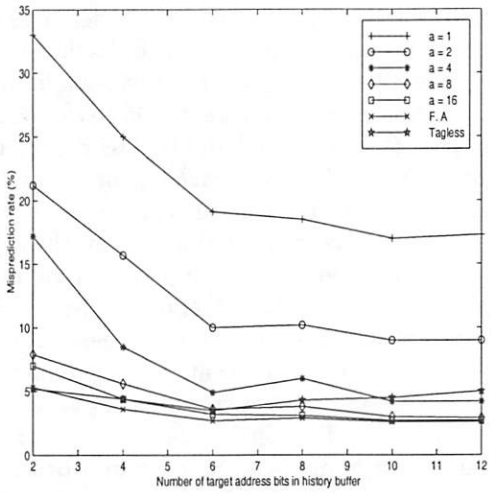


Fig 13: Comparison of tagged and tagless target buffers using *javac*. The size of the tagged and tagless target buffers were 2K and 4K respectively. The XOR hashing scheme with global path history were used for all cases.

Figures 15 and 16 show the variation in misprediction rates for the various buffer sizes. A tagged target buffer requires additional area overhead for maintaining the tags as compared to a tagless target buffer. Hence, a tagless target buffer can have more number of entries corresponding to the same implementation cost. It can be observed that an associative tagged target buffer with 8 or more entries per set outperforms the tagless buffer. It can also be observed that the increase in buffer size reduces the conflict misses significantly for the tagless and tagged buffers with associativity less than or equal to 4. The tagged buffers with associativity greater than 4 do not benefit much from increase in buffer size since the higher number of entries per set already takes care of most of the conflict misses. Due to the increase in access and cycle times associated with the higher associativities [21], the area overhead of the tags in the tagged buffers and the small difference in the misprediction rates between tagless and tagged buffers with large associativities ($a > 4$), the tagless target buffer may be a better choice in many cases.

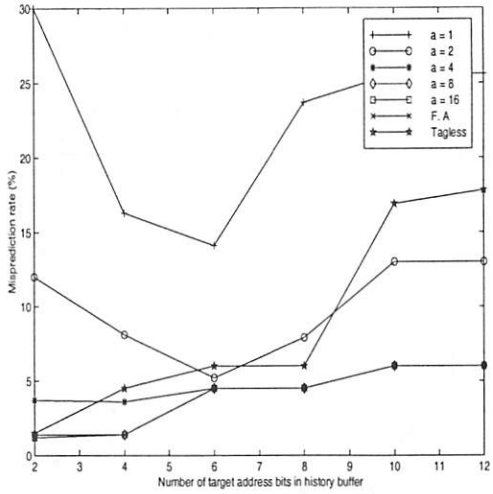


Fig 14: Comparison of tagged and tagless target buffers using *richards*. The size of the tagged and tagless target buffers were 2K and 4K respectively. The XOR hashing scheme with global path history were used for all cases.

6 Conclusion

The effectiveness of using path history to predict the target addresses of indirect branches due to virtual method invocations used in Java applications was investigated. The influence of the various parameters such as number of history buffers, path length, hashing function and the structure of the target buffers on the misprediction rates was investigated. The XOR hashing scheme with a global path history and a 2-bit update pol-

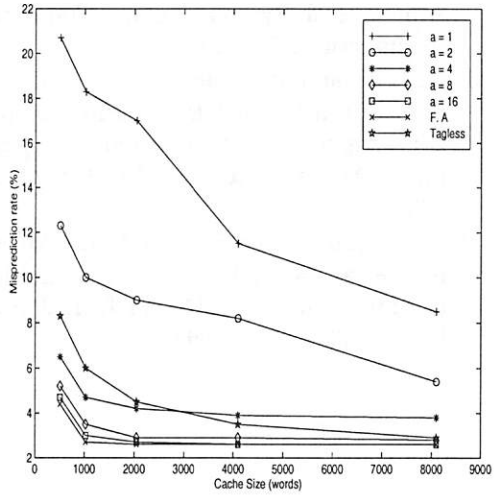


Fig 15: Comparison of tagged and tagless target buffers using *javac* with variation in target buffer size. A path length of 2 was used.

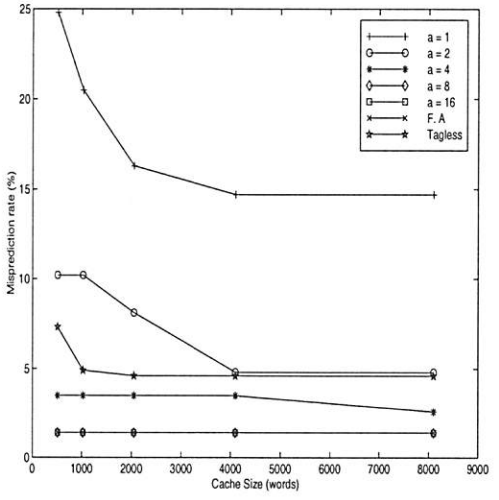


Fig 16: Comparison of tagged and tagless target buffers using *richards* with variation in target buffer size. A path length of 4 was used.

icy performed the best for almost all configurations. Also, it was found that the tagless target buffers achieve a prediction rate as good as the tagged buffers without suffering from the area overhead for tags and the increased access times associated with the associative buffers. Using the branch target buffer based predictor with an 8K buffer, misprediction rates of 4.9% and 23.4% were obtained for the *javac* and *richards* benchmarks respectively. The misprediction rates reduce to 3.6% and 2.4% for the two benchmarks using the proposed path history based predictor. The results show that the design of micro-architectural features such as the branch predictor will influence the execution speed of Java code.

References

[1] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.

[2] A. Krall and R. Grafl, "CACAO - a 64 bit JavaVM just-in-time compiler", Concurrency: Practice and Experience, 9(11):1017-1030, 1997.

[3] B. Venners, "Under the hood: Java's garbage-collected heap", http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html.

[4] M. Wolckzo, "Benchmarking Java with Richards and DeltaBlue", Sun Microsystems. http://www.sunlabs.com/people/mario/java_benchmarking/index.html

[5] U. Holzle, C. Chambers, and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches" Proceedings of ECOOP '91.

[6] J. E. Smith, "A study of branch prediction strategies", Proc. 8th Annual Intl Symposium on Computer Architecture, pp. 135-148, 1981.

[7] T. Yeh and Y. N. Patt, " Two-level adaptive branch prediction", Proc. of the 24th ACM/IEEE Intl Symposium on Microarchitecture, pp 51-61, 1991.

[8] R. Nair, "Dynamic path-based branch correlation", Proc. of the

28th ACM/IEEE Intl Symposium on Microarchitecture, pp 15-23, 1995.

[9] C. A. Hsieh et. al., " A study of cache and branch performance issues with running Java on current hardware platforms", Proc. of COMPCON, Feb 1997, pp. 211-216.

[10] T. H. Romer et. al., "The Structure and Performance of Interpreters", Proceedings of ASPLOS VII, 1996, pp. 150-159.

[11] D. Griswold, "Breaking the speed barrier: the future of Java performance", JavaOne Worldwide Java Developer Conference, 1997.

[12] T. R. Halfhill, Intel's P6, Byte Magazine, April 1995. http://www.byte.com/art/9504/sec7/art1.htm

[13] N. Vijaykrishnan, N. Ranganathan and R. Gadekarla, "Object-Oriented architectural support for a Java processor architecture", Proc. of the 12th European Conference on Object-Oriented Programming, July 1998.

[14] J. A. Dean, Whole-Program optimization of object-oriented languages, Ph.D Thesis, University of Washington, 1996.

[15] J. Vitek, "Compact dispatch tables for dynamically typed programming languages", Object Applications, ed. D. Tsichitzis, University of Geneva, Centre Universitaire d'Informatique, Aug. 1996.

[16] C. Young, N. Gloy and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction", Proc. of the 22nd Annual Intl Symposium on Computer Architecture, June 1995.

[17] P. Y. Chang, E. Hao and Y. Patt, "Target prediction for indirect jumps", Proc. of the 24th Annual Intl Symposium on Computer Architecture, 1997, pp. 274-283.

[18] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs", Proc. of the 6th Intl Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[19] J. Emer and N. Gloy, "A language for describing predictors and its application to automatic synthesis", Proc. of the 24th Annual Intl Symposium on Computer Architecture, July 1997.

[20] K. Dreisen and U. Holzle, "Accurate indirect branch prediction", Proc. of the 25th Annual Intl Symposium on Computer Architecture, pp. 167-178, June 1998.

[21] N. P. Jouppi and S. J. E. Wilton, "An enhanced access and cycle time model for on-chip caches", DEC- WRL Technical Report, 93.5, July 1994.

Comprehensive Profiling Support in the Java™ Virtual Machine

Sheng Liang Deepa Viswanathan
Sun Microsystems Inc.
901 San Antonio Road, CUP02-302
Palo Alto, CA 94303
{sheng.liang,deepa.viswanathan}@eng.sun.com

Abstract

Existing profilers for Java applications typically rely on custom instrumentation in the Java virtual machine, and measure only limited types of resource consumption. Garbage collection and multi-threading pose additional challenges to profiler design and implementation.

In this paper we discuss a general-purpose, portable, and extensible approach for obtaining comprehensive profiling information from the Java virtual machine. Profilers based on this framework can uncover CPU usage hot spots, heavy memory allocation sites, unnecessary object retention, contended monitors, and thread deadlocks. In addition, we discuss a novel algorithm for thread-aware statistical CPU time profiling, a heap profiling technique independent of the garbage collection implementation, and support for interactive profiling with minimum overhead.

1 Introduction

Profiling [14] is an important step in software development. We use the term profiling to mean, in a broad sense, the ability to monitor and trace events that occur during run time, the ability to track the cost of these events, as well as the ability to attribute the cost of the events to specific parts of the program. For example, a profiler may provide information about what portion of the program consumes the most amount of CPU time, or about what portion of the program allocates the most amount of memory.

This paper is mainly concerned with profilers that provide information to programmers, as opposed to profilers that feedback to the compiler or run-time system. Although the fundamental principles of profiling are the same, there are different requirements in designing these two kinds of profilers. For example, a profiler that sends feedback to the run-time system must incur as little overhead as possible so that it does not slow down program execution. A profiler that constructs the complete call graph, on the other hand, may be permitted to slow down the program execution significantly.

This paper discusses techniques for profiling support in the Java virtual machine [17]. Java applications are written in the Java programming language [10], and compiled into machine-independent binary class files, which can then be executed on any compatible implementation of the Java virtual machine. The Java virtual machine is a multi-threaded and garbage-collected execution environment that generates various events of interest for the profiler. For example:

- The profiler may measure the amount of CPU time consumed by a given method in a given class. In order to pinpoint the exact cause of inefficiency, the profiler may need to isolate the total CPU time of a method `A.f` called from another method `B.g`, and ignore all other calls to `A.f`. Similarly, the profiler may only want to measure the cost of executing a method in a particular thread.
- The profiler may inform the programmer why there is excessive creation of object instances that belong to a given class. The programmer may want to know, for example, that many instances of class `D` are allocated in method `C.h`. More specifically, it is also useful to know that majority of these allocations occur when `B.g` calls `C.h`, *and* only when `A.f` calls `B.g`.
- The profiler may show why a certain object is not being garbage collected. The programmer may want to know, for example, that an instance of class `C` is not garbage collected because it is referred to by an instance of class `D`, which is then referred to by a local variable in an active stack frame of method `B.g`.
- The profiler may identify the monitors that are con-

tended by multiple threads. It is useful to know, for example, that two threads, T_1 and T_2, repeatedly contend to enter the monitor associated with an instance of class C.

- The profiler may inform the programmer what causes a given class to be loaded. Class loading not only takes time, but also consumes memory resources in the Java virtual machine. Knowing the exact reason that a class is loaded, the programmer can optimize the code to reduce memory usage.

The first contribution of this paper is to present a general-purpose, extensible, and portable Java virtual machine profiling architecture. Existing profilers typically rely on custom instrumentation in the Java virtual machine and measure limited types of resource consumption. In contrast, our framework relies on an interface that provides comprehensive support for profilers that can be built independent of the Java virtual machine. A profiler can obtain information about CPU usage hot spots, heavy memory allocation sites, unnecessary object retention, monitor contention, and thread deadlocks. Both code instrumentation and statistical sampling are supported. Adding new features typically requires introducing new event types, and does not require changes to the profiling interface itself. The profiling interface is portable. It is not dependent on the internal implementation of the Java virtual machine. For example, the heap profiling support is independent of the garbage collection implementation, and can present useful information for a wide range of garbage collection algorithms. The benefit of this approach is obvious. Tools vendors can ship profilers that work with any virtual machine that implements the interface. Equivalently, users of a Java virtual machine can easily take advantage of the profilers available from different tools vendors.

The second contribution of this paper is to introduce an algorithm that obtains accurate CPU-time profiles in a multi-threaded execution environment with minimum overhead. It is a standard technique to perform statistical CPU time profiling by periodically sampling the running program. What is less known, however, is how to obtain accurate per-thread CPU time usage on the majority of operating systems that do not provide access to the thread scheduler or a high-resolution per-thread CPU timer clock. In these cases, it is difficult to attribute elapsed time to threads that are actually running, as opposed to threads that are blocked, for example, in an I/O operation. Our solution is to determine whether a thread has run in a sampling interval by comparing the check sum of its register sets. To our knowledge, this is the most portable technique for obtaining thread-aware CPU-time profiles on modern operating systems.

The third contribution is to demonstrate how our approach supports interactive profiling with minimum overhead. Users can selectively enable or disable different types of profiling while the application is running. This is achieved with very low space and time overhead. Neither the virtual machine, nor the profiler need to accumulate large amounts of trace data. The Java virtual machine incurs only a test and branch overhead for a disabled profiling event. Most events occur in code paths that can tolerate the overhead of an added check. As a result, the Java virtual machine can be deployed with profiling support in place.

We have implemented all the techniques discussed in this paper in the Java Development Kit (JDK) 1.2 [15]. Numerous tool vendors have already built profiling front-ends that rely on the comprehensive profiling support built into the JDK 1.2 virtual machine.

We will begin by introducing the general-purpose profiling architecture, before we discuss the underlying techniques in detail. We assume the reader is familiar with the basic concepts in the Java programming language [10] and the Java virtual machine [17].

2 Profiling Architecture

The key component of our profiling architecture is a general-purpose profiling interface between the Java virtual machine and the front-end responsible for presenting the profiling information. A profiling interface, as opposed to direct profiling support in the virtual machine implementation, offers two main advantages:

First, profilers can present profiling information in different forms. For example, one profiler may simply record events that occur in the virtual machine in a trace file. Alternatively, another profiler may receive input from the user and display the requested information interactively.

Second, the same profiler can work with different virtual machine implementations, as long as they all support the same profiling interface. This allows tool vendors and virtual machine vendors to leverage each other's products effectively.

A profiling interface, while providing flexibility, also has potential shortcomings. On one hand, profiler front-ends may be interested in a diverse set of events that occur in the virtual machine. On the other hand, virtual machine

implementations from different vendors may be different enough that it is impossible to expose all the interesting events through a general-purpose interface.

The contribution of our work is to reconcile these differences. We have designed a general-purpose Java Virtual Machine Profiler Interface (JVMPI) that is efficient and powerful enough to suit the needs of a wide variety of virtual machine implementations and profiler front-ends.

Figure 1 illustrates the overall architecture. The JVMPI is a binary function-call interface between the Java virtual machine and a *profiler agent* that runs in the same process. The profiler agent is responsible for the communication between the Java virtual machine and the profiler front-end. Note that although the profiler agent runs in the same process as the virtual machine, the profiler front-end typically resides in a different process, or even on a different machine. The reason for the separation of the profiler front-end is to prevent the profiler front-end from interfering with the application. Process-level separation ensures that resources consumed by the profiler front-end does not get attributed to the profiled application. Our experience shows that it is possible to write profiler agents that delegate resource-intensive tasks to the profiler front-end, so that running the profiler agent in the same process as the virtual machine does not overly distort the profiling information.

We will introduce some of the features of the Java virtual machine profiling interface in the remainder of this section, and discuss how such features are supported by the Java virtual machine in later sections.

2.1 Java Virtual Machine Profiler Interface

Figure 1 illustrates the role of the JVMPI in the overall profiler architecture. The JVMPI is a two-way function call interface between the Java virtual machine and the profiler agent.

The profiler agent is typically implemented as a dynamically-loaded library. The virtual machine makes function calls to inform the profiler agent about various events that occur during the execution of the Java application. The agent in turn receives profiling events, and calls back into the Java virtual machine to accomplish one the the following tasks:

- The agent may disable and enable certain type of events sent through the JVMPI, based on the needs of the profiler front-end.
- The agent may request more information in response to particular events. For example, after the agent receives a JVMPI event, it can make a JVMPI function call to find out the stack trace for the current thread, so that the profiler front-end can inform the user about the program execution context that led to this JVMPI event.

Using function calls is a good approach to an efficient binary interface between the profiler agent and different virtual machine implementations. Sending profiling events through function calls is somewhat slower than directly instrumenting the virtual machine to gather specific profiling information. As we will see, however, majority of the profiling events are sent in situations where we can tolerate the additional cost of a function call.

JVMPI events are data structures consisting of an integer indicating the type of the event, the identifier of the thread in which the event occurred, followed by information specific to the event. To illustrate, we list the definition of the `JVMPI_Event` structure and one of its variants `gc_info` below. The `gc_info` variant records information about an invocation of the garbage collector. The event-specific information indicates the number of live objects, total space used by live objects, and the total heap size.

```
typedef struct {
    jint event_type;
    JNIEnv *thread_id;
    ...
    union {
        ...
        struct {
            jlong used_objects;
            jlong used_object_space;
            jlong total_object_space;
        } gc_info;
        ...
    } u;
} JVMPI_Event;
```

Additional details of the JVMPI can be found in the documentation that is shipped with the JDK 1.2 release [15].

2.2 The HPROF Agent

To illustrate the power of the JVMPI and show how it may be utilized, we describe some of the features in the HPROF agent, a simple profiler agent shipped with JDK 1.2. The HPROF agent is a dynamically-linked library

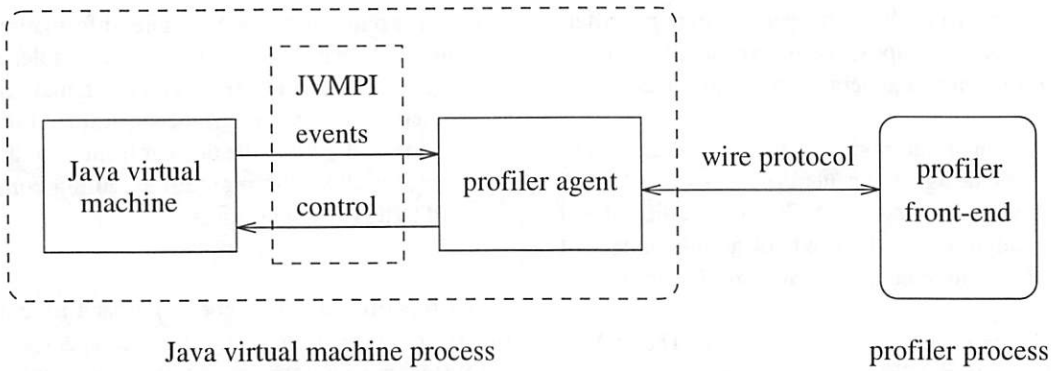


Figure 1: Profiler Architecture

shipped with JDK 1.2. It interacts with the JVMPI and presents profiling information either to the user directly or through profiler front-ends.

We can invoke the HPROF agent by passing a special option to the Java virtual machine:

```
java -Xrunhprof ProgName
```

`ProgName` is the name of a Java application. Note that we pass the `-Xrunhprof` option to `java`, the optimized version of the Java virtual machine. We need not rely on a specially instrumented version of the virtual machine to support profiling.

Depending on the type of profiling requested, HPROF instructs the virtual machine to send it the relevant profiling events. It gathers the event data into profiling information and outputs the result by default to a file. For example, the following command obtains the heap allocation profile for running a program:

```
java -Xrunhprof:heap=sites ProgName
```

Figure 2 contains the heap allocation profile generated by running the Java compiler (`javac`) on a set of input files. We only show parts of the profiler output here due to the lack of space. A crucial piece of information in heap profile is the amount of allocation that occurs in various parts of the program. The `SITES` record above tells us that 9.18% of live objects are character arrays. Note that the amount of live data is only a fraction of the total allocation that has occurred at a given site; the rest has been garbage collected.

A good way to relate allocation sites to the source code is to record the dynamic stack traces that led to the heap allocation. Figure 3 shows another part of the profiler output that illustrates the stack traces referred to by the four allocation sites presented in Figure 2.

Each frame in the stack trace contains class name, method name, source file name, and the line number. The user can set the maximum number of frames collected by the HPROF agent. The default limit is 4. Stack traces reveal not only which methods performed heap allocation, but also which methods were ultimately responsible for making calls that resulted in memory allocation. For example, in the heap profile above, instances of the same `java/util/Hashtable$Entry` class are allocated in traces 1091 and 1264, each originated from different methods.

The HPROF agent has built-in support for profiling CPU usage. For example, Figure 4 is part of the generated output after the HPROF agent performs sampling-based CPU time profiling on the `javac` compiler.

The HPROF agent periodically samples the stack of all running threads to record the most frequently active stack traces. The `count` field above indicates how many times a particular stack trace was found to be active. These stack traces correspond to the CPU usage hot spots in the application.

The HPROF agent can also report complete heap dumps and monitor contention information. Due to the lack of space, we will not list more examples of how the HPROF agent presents the information obtained through the profiling interface. However, we are ready to explain the details of how various profiling interface features are supported in the virtual machine.

```
SITES BEGIN (ordered by live bytes) Wed Oct  7 11:38:10 1998
        percent          live        alloc'ed     stack class
rank  self  accum    bytes objs    bytes   objs trace name
   1 9.18%  9.18%   149224 5916 1984600 129884  1073 char []
   2 7.28% 16.45%   118320 5916  118320   5916  1090 sun/tools/java/Identifier
   3 7.28% 23.73%   118320 5916  118320   5916  1091 java/util/Hashtable$Entry
   ...
   7 3.39% 41.42%    55180 2759   55180   2759  1264 java/util/Hashtable$Entry
   ...
SITES END
```

Figure 2: HPROF Heap Allocation Profile

```
THREAD START (obj=1d6b20, id = 1, name="main", group="main")
...
TRACE 1073: (thread=1)
        java/lang/String.<init>(String.java:244)
        sun/tools/java/Scanner.bufferString(Scanner.java:143)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)

TRACE 1090: (thread=1)
        sun/tools/java/Identifier.lookup(Identifier.java:106)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)
        sun/tools/java/Scanner.scan(Scanner.java:971)

TRACE 1091: (thread=1)
        java/util/Hashtable.put(Hashtable.java:405)
        sun/tools/java/Identifier.lookup(Identifier.java:106)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)

TRACE 1264: (thread=1)
        java/util/Hashtable.put(Hashtable.java:405)
        sun/tools/java/Type.<init>(Type.java:90)
        sun/tools/java/MethodType.<init>(MethodType.java:42)
        sun/tools/java/Type.tMethod(Type.java:274)
```

Figure 3: HPROF Stack Traces

```
CPU SAMPLES BEGIN (total = 252378) Wed Oct 07 13:30:10 1998
rank   self  accum   count trace method
   1  4.96%  4.96%   12514   303 sun/io/ByteToCharSingleByte.convert
   2  3.18%  8.14%    8022   306 java/lang/String.charAt
   3  1.91% 10.05%    4828   301 sun/tools/java/ScannerInputReader.<init>
   4  1.80% 11.85%    4545   305 sun/io/ByteToCharSingleByte.getUnicode
   5  1.50% 13.35%    3783   304 sun/io/ByteToCharSingleByte.getUnicode
   6  1.30% 14.65%    3280   336 sun/tools/java/ScannerInputReader.read
   7  1.13% 15.78%    2864   404 sun/io/ByteToCharSingleByte.convert
   8  1.11% 16.89%    2800   307 java/lang/String.length
   9  1.00% 17.89%    2516  4028 java/lang/Integer.toString
  10  0.95% 18.84%    2403   162 java/lang/System.arraycopy
...
CPU SAMPLES END
```

Figure 4: HPROF Profile of CPU Usage Hot Spots

3 CPU Time Profiling

A CPU time profiler collects data about how much CPU time is spent in different parts of the program. Equipped with this information, programmers can find ways to reduce the total execution time.

3.1 Design Choices

We considered the following design choices when building the support for CPU time profilers: the granularity of profiling information and whether to use statistical sampling or code instrumentation.

3.1.1 Granularity

Shall we present information at the method call level, or at a finer granularity such as basic blocks or different execution paths inside a method? Based on our experience with tuning Java applications, we believe that there is little reason to attribute cost to a finer granularity than methods. Programmers typically have a good understanding of cost distribution inside a method; methods in Java applications tend to be smaller than, for example, C/C++ functions.

It is not enough to report a flat profile consisting only of the portion of time in individual methods. If, for example, the profiler reports that a program spends a significant portion of time in the `String.getBytes` method, how do we know which part of our program indirectly contributed to invoking this method, if the program does not call this method directly?

A good way to attribute profiling information to Java applications is to report the dynamic stack traces that lead to the resource consumption. Dynamic stack traces become less informative in some languages where it is hard to associate stack frames with source language constructs, such as when anonymous functions are involved. Fortunately, anonymous inner classes in the Java programming language are represented by classes with informative names at run time.

3.1.2 Statistical Sampling vs. Code Instrumentation

There are two ways to obtain profiling information: either statistical sampling or code instrumentation. Statistical sampling is less disruptive to program execution, but cannot provide completely accurate information. Code instrumentation, on the other hand, may be more disruptive, but allows the profiler to record all the events it is interested in. Specifically in CPU time profiling, statistical sampling may reveal, for example, the relative percentage of time spent in frequently-called methods, whereas code instrumentation can report the exact number of time each method is invoked.

Our framework supports both statistical sampling and code instrumentation. Through the JVMPI, the profiler agent can periodically sample the stack of all running threads, thus discovering the most frequently active stack traces. Alternatively, the profiler agent may ask the virtual machine to send events on entering and exiting methods. Naturally the latter approach introduces additional C function call overhead to each profiled method.

A less disruptive way to implement code instrumentation is to inject profiling code directly into the profiled program. This type of code instrumentation is easier on the Java platform than on traditional CPUs, because there is a standard class file format. The JVMPI allows the profiler agent to instrument every class file before it is loaded by the virtual machine. The profiler agent may, for example, insert custom byte code sequence that records method invocations, control flow among the basic blocks, or other operations (such as object creation or monitor operations) performed inside the method body. When the profiler agent changes the content of a class file, it must ensure that the resulting class file is still valid according to the Java virtual machine specification.

3.2 Thread-Aware CPU Time Sampling

The Java virtual machine is a multi-threaded execution environment. One difficulty in building CPU time profilers for such systems is how to properly attribute CPU time to each thread, so that the time spent in a method is accounted only when the method actually runs on the CPU, not when it is unscheduled and waiting to run. The basic CPU time sampling algorithm is as follows:

```
while (true) {
 - sleep for a short interval;
 - suspend all threads;
 - record the stack traces of all threads
   that have run in the last interval;
 - attribute a cost unit to these stack
   traces;
 - resume all threads;
}
```

The profiler needs to suspend the thread while collecting its stack trace, otherwise a running thread may change the stack frames as the stack trace is being collected.

The main difficulty in the above scheme is how to determine whether a thread has run in the last sampling interval. We should not attribute cost units to threads that are waiting for an I/O operation, or waiting to be scheduled in the last sampling interval. Ideally, this problem would be solved if the scheduler could inform the profiler the exact time interval in which a thread is running, or if the profiler could find out the amount of CPU time a thread has consumed at each sampling point.

Unfortunately, modern operating systems such as Windows NT and Solaris neither expose the kernel scheduler nor provide ways to obtain accurate per-thread CPU time. For example, the `GetThreadTimes` call on Windows NT returns per-thread CPU time in 10 millisecond increments, too inaccurate for profiling needs.

Our solution is to determine whether a thread has run in a sampling interval by checking whether its register set has changed. If a thread has run in the last sampling interval, it is almost certain that the contents of the register set have changed.

The information gathered for the purpose of profiling need not be 100% reliable. It is extremely unlikely, however, that a running thread maintains an unchanged register set, which includes such registers as the stack pointer, the program counter, and all general-purpose registers. One pathological example of a running program with a constant register set is the following C code segment, where the program enters into an infinite loop that consists of one instruction:

```
loop: goto loop;
```

In practice, we find that it suffices to compute and record a checksum of a subset of the registers, thus further reducing the overhead of the profiler.

The cost of suspending all threads and collecting their stack traces is roughly proportional of the number of threads running in the virtual machine. A minor enhancement to the sampling algorithm discussed earlier is that we need not suspend and collect stack traces for threads that are blocked on monitors managed by the virtual machine. This significantly reduces the profiling overhead for many multi-threaded programs in which most threads are blocked most of the time. Our experience shows that, for typical programs, the total overhead of our sampling-based CPU time profiler with a sampling interval of 1 millisecond is less than 20%.

4 Heap Profiling

Heap profiling serves a number of purposes: pinpointing the part of program that performs excessive heap allocation, revealing the performance characteristics of the underlying garbage collection algorithm, and detecting the causes of unnecessary object retention.

4.1 Excessive Heap Allocation

Excessive heap allocation leads to performance degradation for two reasons: the cost of the allocation operations themselves, and because the heap is filled up more quickly, the cost of more frequent garbage collections. With the JVMPI, the profiler follows the following steps to pinpoint the part of the program that performs excessive heap allocation:

- Enable the event notification for object allocation, so that the virtual machine issues a function call to the profiler agent when the current thread performs heap allocation.
- Obtain the current stack trace from the virtual machine when object allocation event arrives. The stack trace serves as a good identification of the heap allocation site. The programmer should concentrate on optimizing busy heap allocation sites.
- Enable the event notification for object reclamation, so that the profiler can keep track of how many objects allocated from a given site are being kept live.

4.2 Algorithm-Independent Allocation and Garbage Collection Events

Many memory allocation and garbage collection algorithms are suitable for different Java virtual machine implementations. Mark-and-sweep, copying, generational, and reference counting are some examples. This presents a challenge to designing a comprehensive profiling interface: Is there a set of events that can uniformly handle a wide variety of garbage collection algorithms?

We have designed a set of profiling events that covers all garbage collection algorithms we are currently concerned with. We introduce the abstract notion of an *arena*, in which objects are allocated. The virtual machine issues the following set of events:

- `NEW_ARENA`(arena ID)

- DELETE_ARENA(arena ID)
- NEW_OBJECT(arena ID, object ID, class ID)
- DELETE_OBJECT(object ID)
- MOVE_OBJECT(old arena ID, old object ID, new arena ID, new object ID)

Our notation encodes the event-specific information in a pair of parentheses, immediately following the event type. Let us go through some examples to see how these events may be used with different garbage collection algorithms:

- A mark-and-sweep collector issues NEW_OBJECT events when allocating objects, and issues DELETE_OBJECT events when adding objects to the free list. Only one arena ID is needed.
- A mark-sweep-compact collector additionally issues MOVE_OBJECT events. Again, only one arena is needed, the old and new arena IDs in the MOVE_OBJECT events are the same.
- A standard two-space copying collector creates two arenas. It issues MOVE_OBJECT events during garbage collection, and a DELETE_ARENA event followed by a NEW_ARENA event with the same arena ID to free up all remaining objects in the semi-space.
- A generational collector issues a NEW_ARENA event for each generation. When an object is scavenged from one generation to anther, a MOVE_OBJECT event is issued. All objects in an arena are implicitly freed when DELETE_ARENA event arrives.
- A reference-counting collector issues NEW_OBJECT events when new objects are created, and issues DELETE_OBJECT events when the reference count of an object reaches zero.

In summary, the simple set of heap allocation events support a wide variety of garbage collection algorithms.

4.3 Unnecessary Object Retention

Unnecessary object retention occurs when an object is no longer useful, but being kept alive by another object that is in use. For example, a programmer may insert objects into a global hash table. These objects cannot be garbage collected, as long as any entry in the hash table is useful and the hash table is kept alive.

An effective way to find the causes of unnecessary object retention is to analyze the heap dump. The heap dump contains information about all the garbage collection roots, all live objects, and how objects refer to each other.

Our profiling interface allows the profiler agent to request the entire heap dump, which can in turn be sent to the profiler front-end for further processing and analysis.

An alternative way to track unnecessary object retention is to provide the direct support in the profiling interface for finding all objects that refer to a given object. The advantage of this incremental approach is that it requires less temporary storage than complete heap dumps. The disadvantage is that unlike heap dumps, the incremental approach cannot present a consistent view of all heap objects that are constantly being modified during program execution.

In practice, we do not find the size of heap dumps to be a problem. Typically the majority of the heap space is occupied by primitive arrays. Because there are no internal pointers in primitive arrays, elements of primitive arrays need not be part of the heap dump.

5 Monitor Profiling

Monitors are the fundamental synchronization mechanism in the Java programming language. Programmers are generally concerned with two issues related to monitors: the performance impact of *monitor contention* and the cause of *deadlocks*. With the recent advances in monitor implementation [4] [21], non-contended monitor operations are no longer a performance issue. A non-contended monitor enter operation, for example, takes only 4 machine instructions on the x86 CPUs [21]. In properly tuned programs, vast majority of monitor operations are non-contended. For example, Table 1 shows the ratio of contended monitor operations in a number of programs. The first 8 applications are from the SPECjvm98 benchmark. The last two applications are GUI-rich programs. The monitor contention rate is extremely low in all programs. In fact, all but one program (mtrt) in the SPECjvm98 benchmark suite are single-threaded.

program	# non-contended	# contended	percent contended
compress	14627	0	0.00%
jess	4826524	0	0.00%
raytrace	377921	0	0.00%
db	53417611	0	0.00%
javac	17337221	0	0.00%
mpeg	14546	0	0.00%
mtrt	715233	11	0.002%
jack	11929729	0	0.00%
HotJava	2277113	564	0.02%
SwingSet	1587585	1332	0.08%

Table 1: Monitor Contention Rate of Benchmark Programs

5.1 Monitor Contention

Monitor contention is the primary cause of lack of scalability in multi-processor systems. Monitor contention is typically caused by multiple threads holding global locks too frequently or too long. To detect these scenarios, the profiler may enable the following three types of event notifications:

- A thread waiting to enter a monitor that is already owned by another thread issues a `MONITOR_CONTENDED_ENTER` event. This event indicates possible performance bottlenecks caused by frequently-contended monitors.
- After a thread finishes waiting to enter a monitor and acquires the monitor, it issues a `MONITOR_CONTENDED_ENTERED` event. This event indicates the amount of elapsed time the current thread has been blocked before it enters the monitor.
- When a thread exits a monitor, and discovers that another thread is waiting to enter the same monitor, the current thread issues a `MONITOR_CONTENDED_EXIT` event. This event indicates possible performance problems caused by the current thread holding the monitor for too long.

In all these three cases, overhead of issuing the event is negligible compared to the performance impact of the blocked monitor operation. The profiler agent can obtain the stack trace of the current thread and thus attribute the monitor contention events to the parts of the program responsible for issuing the monitor operations.

5.2 Deadlocks

If every thread is waiting to enter monitors that are owned by another thread, the system runs into a deadlock situation. A thread/monitor dump is what programmers need to find the cause of this kind of deadlocks.[1] A thread/monitor dump includes the following information:

- The stack trace of all threads.
- The owner of each monitor and the list of threads that are waiting to enter the monitor.

To obtain a consistent view of all threads and all monitors, we suspend all threads when collecting thread/monitor dumps. The JDK has historically provided support for thread/monitor dumps triggered by special key sequences (such as Ctrl-Break on Win32). The JVMPI now allows the profiler agent to obtain the same information programmatically.

6 Support for Interactive Low-Overhead Profiling

The profiling support we built into the Java virtual machine achieves the following two desirable goals:

- We must be able to support *interactive profiler front-ends*. An approach that only supports collecting profiling events into trace files does not meet the needs of programmers and tools vendors. The

[1] Deadlocks may also be caused by implicit locking and ordering in libraries and system calls, such as I/O operations.

user must to enable and disable profiling events during program execution in order to pinpoint performance problems in different stages of running an application.

- The profiling support must incur *low overhead* so that programmers can run the application at full speed when profiling events are disabled, and only pay for the overhead of generating the type of events specifically requested by the profiler front-end. An approach that requires the use of a less optimized virtual machine implementation for profiling leads to additional discrepancies between the profiled environment and real-world scenarios.

Because of the low overhead of our approach, we are able to provide full profiling support in the standard deployed version of the Java virtual machine implementation. It is possible to start an application normally, and enable the necessary profiling events later without restarting the application.

6.1 Overhead of Disabled Profiling Events

The need for dynamically enabling and disabling profiling events requires added checks in the code paths that lead to the generation of these events.

Majority of profiling events are issued relatively infrequently. Examples of these types of events are class loading and unloading, thread start and end, garbage collection, and JNI global reference creation and deletion. We can easily support interactive low-overhead profiling by placing checks in the corresponding code paths without having a performance impact in normal program execution.

Heap profiling events, in particular `NEW_OBJECT`, `DELETE_OBJECT`, and `MOVE_OBJECT` introduced in Section 4.2, could be quite frequent. An added check in every object allocation may have a noticeable performance impact in program execution, especially if the check is inserted in the allocation fast path that typically is inlined into the code generated by the Just-In-Time (JIT) compilers. Fortunately, garbage-collected memory systems by definition need to check for possible heap exhaustion conditions in every object allocation, even in the fast path. We can thus enable heap allocation events by forcing every object allocation into the slow path with a false heap exhaustion condition, and check whether heap profiling events have been enabled and whether the heap is really exhausted in the slow path. Because no change to the allocation fast path is needed, object allocation runs in full speed when heap profiling is disabled.

Method enter and exit events are another kind of events that may be generated frequently. They can be easily supported by the JIT compilers that can dynamically patch the generated code and the virtual method dispatch tables.

6.2 The Partial Profiling Problem

A problem that arises when profiler events can be enabled and disabled is that the profiler agent receives incomplete, or partial, profiling information. This has been characterized as the *partial profiling problem* [16]. For example, if the profiler agent enables the thread start and end events after a thread has been started, it will receive an unknown thread ID that has not been defined in any thread start event. Similarly, if the profiler agent enables the class load event after a number of classes have been loaded and a number of instances of these classes have been created, the agent may encounter `NEW_OBJECT` events that contain an unknown class ID.

A straightforward solution is to require the virtual machine to record all profiling events in a trace file, whether or not these events are enabled by the profiler agent. The virtual machine is then able to send the appropriate information for any entities unknown to the profiler agent. This approach is undesirable because of the potentially unlimited size of the trace file and the overhead when profiling events are disabled.

We solve the partial profiling problem based on one observation: The Java virtual machine keeps track of information internally about the valid entities (such as class IDs) that can be sent with profiling events. The virtual machine need not keep track of outdated entities (such as a class that has been loaded and unloaded) because they will not not appear in profiling events. When the profiler agent receives an unknown entity (such as an unknown class ID), the entity is still valid, and thus the agent can immediately obtain all the relevant information from the virtual machine. We introduce a JVMPI function that allows the profiling agent to request information about unknown entities received as part of a profiling event. For example, when the profiler agent encounters an unknown class ID, it may request the virtual machine to send the same information that is contained in a class load event for this class.

Certain entities need to be treated specially by the pro-

filing agent in order to deal with partial profiling information. For example, if the profiling agent disables the `MOVE_OBJECT` event, it must immediately invalidate all object IDs it knows about, because they may be changed by future garbage collections. With the `MOVE_OBJECT` event disabled, the agent can request the virtual machine to send the class information about unknown object IDs. However, such requests must be made only when garbage collection is disabled (by, for example, calling one of the JVMPI functions). Otherwise garbage collection may generate a `MOVE_OBJECT` event asynchronously and invalidate the object ID before the virtual machine obtains the class information for this object ID.

7 Related Work

Extensive work has been done in CPU time profiling. The `gprof` tool [11], for example, is an sample-based profiler that records call graphs, instead of flat profiles. Recent research [7] [18] [19] has improved the performance and accuracy of time profilers based on code instrumentation. Analysis techniques have been developed such that instrumentation code may be inserted with as little run-time overhead as possible [5] [1]. Our sampling-based CPU time profiling uses stack traces to report CPU usage hot-spots, and is the most similar to the technique of call graph profiling [12]. Sansom et al [20] investigated how to properly attribute costs in profiling higher-order lazy functional programs. Appel et al [2] studied how to efficiently instrument code in the presence of code inlining and garbage collection. None of the above work addresses the issues in profiling multi-threaded programs, however.

Issues similar to profiling multi-threaded programs arise in parallel programs [3] [13], where the profiler typically executes concurrently with the program, and can selectively profile parts of the program.

Heap profiling similar to that reported in this paper has been developed for C, Lisp [22], and Modula-3 [9]. To our knowledge, our work is the first that constructs a heap profiling interface that is independent of the underlying garbage collection algorithm.

We have a general-purpose profiling architecture, but sometimes it is also useful to build custom profilers [8] that target specific compiler optimizations.

There have been numerous earlier experiments (for example, [6]) on building interactive profiling tools for Java applications. These approaches are typically based on placing custom instrumentation in the Java virtual machine implementation.

8 Conclusions

We have presented a profiling architecture that provides comprehensive profiling support in the Java virtual machine. The scope of profiling information includes multi-threaded CPU usage hot spots, heap allocation, garbage collection, and monitor contention. Our framework supports interactive profiling, and carries extremely low run-time overhead.

We believe that our work lays a foundation for building advanced profiling tools.

Acknowledgements

We wish to thank a number of colleagues at IBM, in particular Robert Berry and members of the Jinsight team, for numerous comments, proposals, and discussions that led to many improvements in the JVMPI. We also appreciate the comments given by the anonymous reviewers of the COOTS'99 conference.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[2] Andrew W. Appel, Bruce F. Duba, David B. MacQueen, and Andrew P. Tolmach. Profling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, 1988.

[3] Ziya Aral and Ilya Gernter. Non-intrusive and interactive profiling in parasight. In *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems*, pages 21–30, July 1988.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[5] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[6] John J. Barton and John Whaley. A real-time performance visualizer for Java. *Dr. Dobb's Journal*, pages 44–48, March 1998.

[7] Matt Bishop. Profiling under UNIX by patching. *Software—Practice and Experience*, 17(10):729–739, October 1987.

[8] Michal Cierniak and Suresh Srinivas. Java and scientific programming: Portable browsers for performance programming. In *Java for Computational Science and Engineering – Simulation and Modeling II*, June 1997.

[9] David L. Detlefs and Bill Kalsow. Debugging storage management problems in garbage-collected environments. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 69–82, June 1995.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[12] Robert J. Hall and Aaron J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of Summer 1993 USENIX Technical Conference*, pages 1–13, June 1993.

[13] Jonathan M.D. Hill, Stephen A. Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool: a case study in optimizing SQL. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing – PDP'98*, pages 286–294, January 1998.

[14] Dan Ingalls. The execution profile as a programming tool. In R. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[15] Java Software, Sun Microsystems, Inc. *Java Development Kit (JDK) 1.2*, 1998. Available at `http://java.sun.com/products/jdk/1.2`.

[16] The Jinsight team (IBM Corporation). Private communication, July 1998.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[18] Carl Ponder and Richard J. Fateman. Inaccuracies in program profilers. *Software—Practice and Experience*, 18(5):459–467, May 1988.

[19] John F. Reiser and Joseph P. Skudiarek. Program profiling problems, and a solution via machine language rewriting. *ACM SIGPLAN Notices*, 29(1):37–45, January 1994.

[20] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 355–366, New York, January 1995. ACM Press.

[21] Hong Zhang and Sheng Liang. Fast monitor implementation for the Java virtual machine. Submitted for publication.

[22] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for c and lisp programs. In *Proceedings of Summer USENIX'88 Conference Proceedings*, pages 223–237, June 1988.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rate for *The Linux Journal, The Perl Journal, IEEE Concurrency,* and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
Microsoft Research
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
ESM Services, Inc.
Global Networking & Computing, Inc.
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe (UGU)

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-35-9